TURING 图灵程序设计丛书

自制编程语言

[日] 前桥和弥 著
刘卓 徐谦 吴雅明 译

人民邮电出版社
POSTS & TELECOM PRESS

前桥和弥（Maebasi Kazuya）

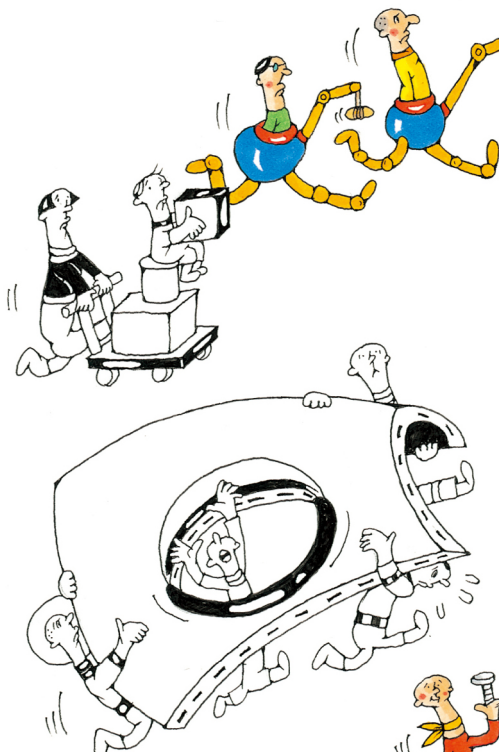
1969年出生，著有《征服C指针》、《彻底掌握C语言》、《Java之谜和陷阱》等。其一针见血的“毒舌”文风和对编程语言深刻的见地受到广大读者的欢迎。

作者主页：http://kmaebashi.com/。

刘卓　2004年开始对日软件开发工作，其间还从事技术及软件工程相关培训工作。自2011年开始从事电力行业产品研发。持续关注企业级应用架构和Web客户端技术。

徐谦　6年技术开发及项目经验，曾以技术工程师身份赴日本工作两年，后归国联合创办互联网公司，现居上海继续创业中。主要从事PHP方向的Web开发。热爱开源，曾向Zend Framework等知名PHP开源项目贡献代码，并于Github自主研发运维EvaThumber等开源项目，获得国内社区认可。乐于分享技术心得，个人技术博客avnpc.com在国内PHP圈小有影响。

吴雅明　13年编程经验，其中7年专注于研发基于Java EE和.NET的开发框架以及基于UML 2.0模型的代码生成工具。目前正带领团队开发云计算PaaS平台及云计算自动化配置部署的系统。译著有《征服C指针》等。



版 权 声 明

（图灵公司感谢李典对本书的审读）

译者序

能翻开这本书的人，想必对编程都有着浓厚的兴趣。大部分编程爱好者都会利用业余时间写一些小程序、开源项目作为消遣，却很少有人会想要自己创造一门编程语言，这是为什么呢？

在翻译本书之前，如果别人问我要不要尝试自制编程语言，我一定会觉得他疯了。因为在潜意识里，我一直认为制作编程语言应该是 C 语言之父丹尼斯·里奇这样的业界大牛才能完成的浩大工程，作为一个普通程序员只要安于本分，用好已有的语言就已经足够了。

在翻译完本书后，我才发现自己真的是大错特错。原来创造一门编程语言，只需要一些 C 语言基础、一些正则表达式知识、加上不断思索的大脑就可以做到。如果你还觉得难以置信，那么就请看看在这本不算厚的书中，作者居然已经创造了两门编程语言，并且都具备高级编程语言的所有特性。

其实一开始的问题已经有了答案：很多看似难如登天的事情，一旦真的下决心去做，你会发现难度并没有想象中那么高，只是我们往往缺少一颗勇于挑战的心罢了。

本书记录了作者一步一步从零创造出编程语言的全过程，作者并不是什么行业精英，而是像你我一样的普通开发者。整本书中也没有用特别复杂的算法或酷炫的编程技巧，但是就凭借着一行行简单朴实的编程语句，作者最终完成了一个普通开发者看来几乎不可能完成的任务。阅读完本书后，除了自制编程语言的知识，我相信读者还能收获到一些更重要的东西。

本书原文讲到了日文编码的知识，为了更好的将内容精髓呈现给读者，我们大胆地将涉及日文编码的部分全部更改为中文编码的知识，译者刘卓还对此编写了很多原创的补充内容，力求能与原书保持同样的水平。如有错误或疏漏，还请读者随时指正。

读完全书后，你会对编程语言的原理和实现方式有一个全面深入的了解，比如你会明白为什么 Java 中 String 类型明明是对象类型却不能改变其内容，C 语言中为什么 `a++ + ++b` 这样看似合理的语句却会报错等。以前

知其然而不知其所以然的问题都会得到答案，这对日后进行更高阶的开发有很大的帮助。

更重要的是，你可以获得自制编程语言的能力，从而可以去做很多以前敢想却没有能力做的事情，比如我现在就在构思能否创造一门以文言文和中国古代文化为基础的编程语言：易经八卦就是天然的二维矩阵，《九章算术》则有不少基础算法……相信读者还会有更加天才有趣的想法出现。如果能运用本书中的知识最终将其实现，那么这将是对翻译工作最好的肯定。

最后，在这里代表其他二位译者一并感谢在翻译过程中给予我们帮助和支持的家人、同事，让这本书最终得以问世。

徐谦

2013 年中秋

前　言

这本书是为那些想独立制作一门编程语言的人而写的。

一听到这个话题，有的人会想：太疯狂了，制作编程语言肯定很有难度吧？有人会怀疑：制作编程语言能有什么用呢？其实这些都是误解。

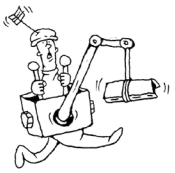
制作编程语言在技术层面上其实并不难，只要掌握一些基础知识即可。而且，制作编程语言对于我们深入理解日常使用的 C、Java、JavaScript 等语言都有帮助。在一些应用程序的内置脚本语言中，我们也经常会因为种种限制从而萌生制作替代语言的想法。因此，自制编程语言并不是少数极客的个人癖好，它对大多数程序员都颇具实用价值。

日本关于制作编程语言的书已经很多了，其中一些还被选定为大学教科书。这些书中常出现有限状态机、NFA、LL(1)、LR(1)、SLA 等专业词汇，同时还大量使用∩、∈等数学符号，对于不熟悉这部分理论知识的人（包括我自己在内）来说非常难以读懂。针对这种现状，本书会偏重实践，避免枯燥的理论。

本书将分别制作两种编程语言：crowbar 与 Diksam。crowbar 是运行分析树的无类型语言，Diksam 是运行字节码的静态类型语言。无论哪种语言，都具备四则运算、变量、条件分支、循环、函数定义、垃圾回收等功能，最终版则可以支持面向对象、异常处理等高级机制。总之，作为现代编程语言所必须具备的功能都基本覆盖了（唯一可能没实现的就是多线程了吧）。所有源代码都提供下载，读者可以一边对照书中的说明一边调试源代码，这样应该不难理解整个程序的运行机制。

当然，要一次实现如此多功能的编程语言，对于初学者而言可能有点吃力，因此本书会详细介绍 crowbar、Diksam 的制作步骤，请放心。

在制作编程语言的过程中，我体会到了一种无法用语言形容的快乐。其实无论在日本或其他地区，世界上还有很多人都在尝试自制编程语言，这正是编程语言不断增加的原因。如果以本书为契机，有朝一日你也向本已混乱的巴比伦之塔再添一门新语言的话，作为本书作者，这将是无上的光荣。

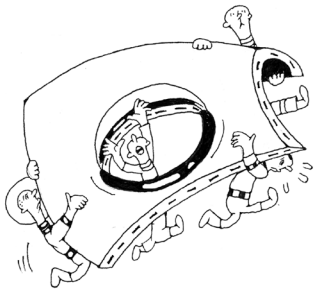


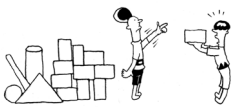
在本书的撰写过程中，得到了很多朋友的帮助与支持：

感谢百忙之中通读原稿并给出很多改进意见的吉田敦、间野健二、藤井壮一、山本将；感谢对本书原型，即网页版“自制编程语言”提出意见的朋友；感谢对博客连载“自制编程语言日记”提出意见的读者朋友，以及实际使用 crowbar 与 Diksam 并提出意见的朋友。最后还要感谢每次对我延迟交稿仍然充满耐心的技术评论社的熊谷裕美子编辑。多亏大家的鼎力支持，本书才终能完成，在此我表示深深的谢意。

2009 年 5 月 7 日 01:06 J.S.T

前桥和弥





目录CONTENTS

第1章 引子 001

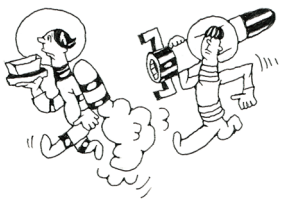
1.1 为什么要制作编程语言 002
1.2 自制编程语言并不是很难 003
1.3 本书的构成与面向读者 004
1.4 用什么语言来制作 006
1.5 要制作怎样的语言 007
1.5.1 要设计怎样的语法 007
1.5.2 要设计怎样的运行方式 009
补充知识 “用户”指的是谁？ 012
补充知识 解释器并不会进行翻译 012
1.6 环境搭建 012
1.6.1 搭建开发环境 012
补充知识 关于bison与flex的安装 014
1.6.2 本书涉及的源代码以及编译器 015

第2章 试做一个计算器 017

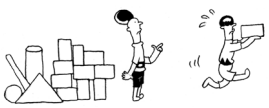
2.1 yacc/lex是什么 018
补充知识 词法分析器与解析器是各自独立的 019
2.2 试做一个计算器 020
2.2.1 lex 021
2.2.2 简单正则表达式讲座 024
2.2.3 yacc 026
2.2.4 生成执行文件 033
2.2.5 理解冲突所代表的含义 034
2.2.6 错误处理 040

2.3 不借助工具编写计算器 041
2.3.1 自制词法分析器 041
补充知识 保留字（关键字）..... 046
补充知识 避免重复包含 047
2.3.2 自制语法分析器 048
补充知识 预读记号的处理 053
2.4 少许理论知识——LL(1)与LALR(1) 054
补充知识 Pascal/C中的语法处理诀窍 056
2.5 习题：扩展计算器 056
2.5.1 让计算器支持括号 056
2.5.2 让计算器支持负数 058

第3章 制作无类型语言crowbar 061



3.1 制作crowbar ver.0.1语言的基础部分 062
3.1.1 crowbar是什么 062
3.1.2 程序的结构 063
3.1.3 数据类型 064
3.1.4 变量 064
补充知识 初次赋值兼做变量声明的理由 066
补充说明 各种语言的全局变量处理 067
3.1.5 语句与结构控制 067
补充知识 elif、elsif、elseif的选择 068
3.1.6 语句与运算符 069
3.1.7 内置函数 069
3.1.8 让crowbar支持C语言调用 070
3.1.9 从crowbar中调用C语言（内置函数的编写）..... 071
3.2 预先准备 071



3.2.1 模块与命名规则 072
3.2.2 内存管理模块MEM 073
补充知识 valgrind 075
补充知识 富翁式编程 075
补充知识 符号表与扣留操作 076
3.2.3 调试模块DBG 076
3.3 crowbar ver.0.1的实现 077
3.3.1 crowbar的解释器——CRB_Interpreter 077
补充知识 不完全类型 080
3.3.2 词法分析——crowbar.l 081
补充知识 静态变量的许可范围 084
3.3.3 分析树的构建——crowbar.y与create.c 085
3.3.4 常量折叠 089
3.3.5 错误信息 089
补充知识 关于crowbar中使用的枚举型定义 091
3.3.6 运行——execute.c 092
3.3.7 表达式评估——eval.c 096
3.3.8 值——CRB_Value 104
3.3.9 原生指针型 105
3.3.10 变量 106
3.3.11 字符串与垃圾回收机制——string_pool.c 108
3.3.12 编译与运行 110

第4章 数组和mark–sweep垃圾回收器 113

4.1 crowbar ver.0.2 114
4.1.1 crowbar的数组 114
4.1.2 访问数组元素 115

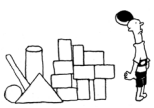
4.1.3 数组是一种引用类型 116
补充知识 “数组的数组”和多维数组 116
4.1.4 为数组添加元素 118
4.1.5 增加(模拟)函数调用功能 118
4.1.6 其他细节 118
4.2 制作mark-sweep GC 119
4.2.1 引用数据类型的结构 119
4.2.2 mark-sweep GC 121
补充知识 引用和immutable 123
4.2.3 crowbar栈 124
4.2.4 其他根 127
4.2.5 原生函数的形式参数 128
4.3 实现GC本身 129
4.3.1 对象的管理方法 129
4.3.2 GC何时启动 129
4.3.3 sweep阶段 132
补充知识 GC现存的问题 133
补充知识 Coping GC 134
4.4 其他修改 136
4.4.1 修改语法 136
4.4.2 函数的模拟 137
4.4.3 左值的处理 139
4.4.4 创建数组和原生函数的书写方法 142
4.4.5 原生指针类型的修改 144

第5章 中文支持和Unicode 147
5.1 中文支持策略和基础知识 148

5.1.1 现存问题 148
5.1.2 宽字符（双字节）串和多字节字符串 149
补充知识 wchar_t 肯定能表示 1 个字符吗？ 150
5.1.3 多字节字符/宽字符之间的转换函数群 150
5.2 Unicode 153
5.2.1 Unicode 的历史 153
5.2.2 Unicode 的编码方式 154
补充知识 Unicode 可以固定（字节）长度吗？ 156
5.3 crowbar book_ver.0.3 的实现 156
5.3.1 要实现到什么程度？ 156
5.3.2 发起转换的时机 157
5.3.3 关于区域设置 158
5.3.4 解决 0x5C 问题 158
补充知识 失败的#ifdef 160
5.3.5 应该是什么样子 160
补充知识 还可以是别的样子——Code Set Independent 161

第 6 章 制作静态类型的语言 Diksam 163
6.1 制作 Diksam Ver 0.1 语言的基本部分 164
6.1.1 Diksam 的运行状态 164
6.1.2 什么是 Diksam 165
6.1.3 程序结构 165
6.1.4 数据类型 166
6.1.5 变量 166
6.1.6 语句和流程控制 167
6.1.7 表达式 167
6.1.8 内建函数 168

6.1.9 其他 168
6.2 什么是静态的/执行字节码的语言 169
6.2.1 静态类型的语言 169
6.2.2 什么是字节码 169
6.2.3 将表达式转换为字节码 170
6.2.4 将控制结构转换为字节码 173
6.2.5 函数的实现 173
6.3 Diksam ver.0.1的实现——编译篇 175
6.3.1 目录结构 175
6.3.2 编译的概要 176
6.3.3 构建分析树（create.c） 176
6.3.4 修正分析树（fix_tree.c） 179
6.3.5 Diksam的运行形式——DVM_Executable 185
6.3.6 常量池 186
补充知识 YARV的情况 187
6.3.7 全局变量 188
6.3.8 函数 189
6.3.9 顶层结构的字节码 189
6.3.10 行号对应表 190
6.3.11 栈的需要量 190
6.3.12 生成字节码（generate.c） 191
6.3.13 生成实际的编码 193
6.4 Diksam虚拟机 197
6.4.1 加载/链接DVM_Executable到DVM 200
6.4.2 执行——巨大的switch case 202
6.4.3 函数调用 204



第7章 为Diksam引入数组.....207

7.1 Diksam中数组的设计.....208

7.1.1 声明数组类型的变量.....208

7.1.2 数组常量.....209

补充知识 D语言的数组.....210

7.2 修改编译器.....210

7.2.1 数组的语法规则.....210

7.2.2 TypeSpecifier结构体.....212

7.3 修改DVM.....213

7.3.1 增加指令.....213

补充知识 创建Java的数组常量.....215

补充知识 C语言中数组的初始化.....217

7.3.2 对象.....217

补充知识 ArrayStoreException.....218

7.3.3 增加null.....219

7.3.4 哎！还缺点什么吧？.....219

第8章 将类引入Diksam.....221

8.1 分割源文件.....222

8.1.1 包和分割源代码.....222

补充知识 #include、文件名、行号.....225

8.1.2 DVM_ExecutableList.....225

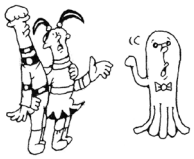
8.1.3 ExecutableEntry.....226

8.1.4 分开编译源代码.....227

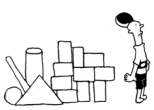
8.1.5 加载和再链接.....230

补充知识 动态加载时的编译器.....233

8.2 设计Diksam中的类.....233



8.2.1 超简单的面向对象入门.....233
8.2.2 类的定义和实例创建.....237
8.2.3 继承.....239
8.2.4 关于接口.....241
8.2.5 编译与接口.....242
8.2.6 Diksam怎么会设计成这样?.....243
8.2.7 数组和字符串的方法.....245
8.2.8 检查类的类型.....246
8.2.9 向下转型.....246
8.3 关于类的实现——继承和多态.....247
8.3.1 字段的内存布局.....247
8.3.2 多态——以单继承为前提.....249
8.3.3 多继承——C++.....250
8.3.4 Diksam的多继承.....252
补充知识 无类型语言中的继承.....254
8.3.5 重写的条件.....254
8.4 关于类的实现.....256
8.4.1 语法规则.....256
8.4.2 编译时的数据结构.....258
8.4.3 DVM_Executable中的数据结构.....260
8.4.4 与类有关的指令.....262
补充知识 方法调用、括号和方法指针.....263
8.4.5 方法调用.....264
8.4.6 super.....266
8.4.7 类的链接.....266
8.4.8 实现数组和字符串的方法.....267
8.4.9 类型检查和向下转型.....267
补充知识 对象终结器（finalizer）和析构函数（destructor）.....268



第9章 应用篇 271

9.1 为crowbar引入对象和闭包 272

9.1.1 crowbar的对象 272

9.1.2 对象实现 273

9.1.3 闭包 274

9.1.4 方法 276

9.1.5 闭包的实现 278

9.1.6 试着跟踪程序实际执行时的轨迹 281

9.1.7 闭包的语法规则 284

9.1.8 普通函数 284

9.1.9 模拟方法（修改版） 285

9.1.10 基于原型的面向对象 286

9.2 异常处理机制 286

9.2.1 为crowbar引入异常 286

9.2.2 setjmp()/longjmp() 289

补充知识 Java和C#异常处理的不同 293

9.2.3 为Diksam引入异常 295

补充知识 catch的编写方法 296

9.2.4 异常的数据结构 297

9.2.5 异常处理时生成的字节码 299

9.2.6 受查异常 301

补充知识 受查异常的是与非 303

补充知识 异常处理本身的是与非 304

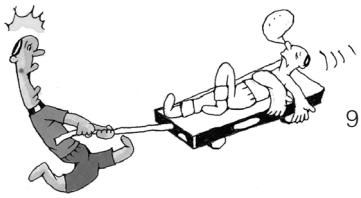
9.3 构建脚本 305

9.3.1 基本思路 306

9.3.2 YY_INPUT 307

9.3.3 Diksam的构建脚本 308

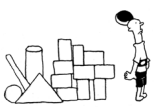
9.3.4 三次加载/链接 308

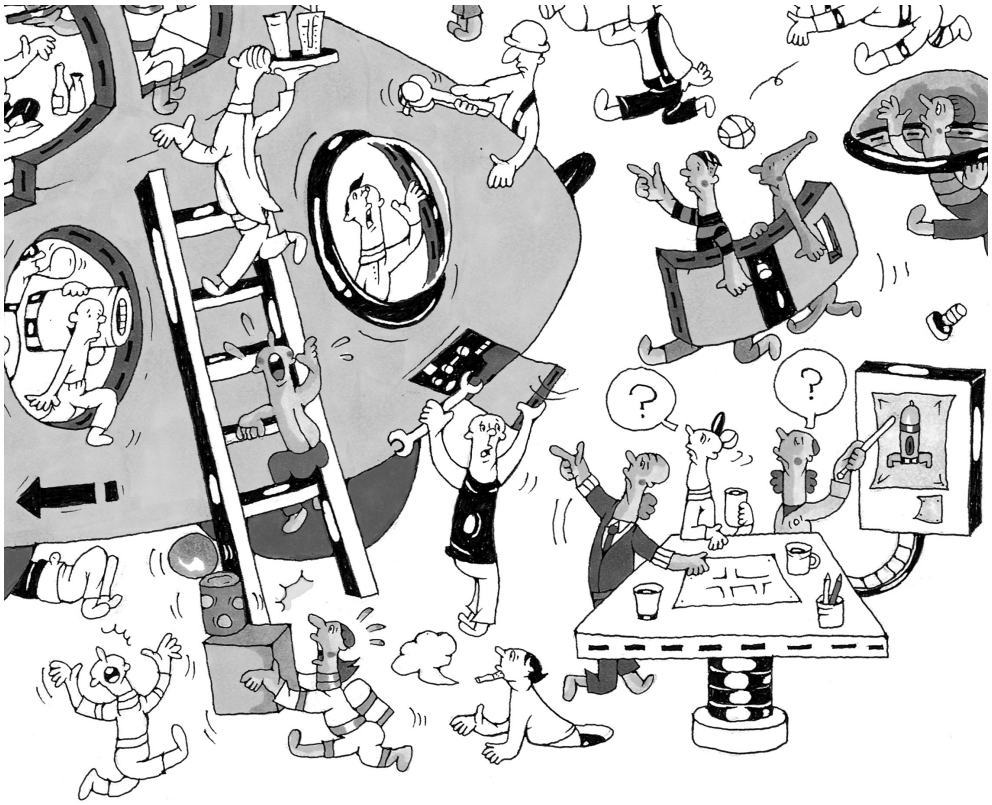


9.4 为crowbar引入鬼车 309
9.4.1 关于“鬼车” 309
9.4.2 正则表达式常量 310
9.4.3 正则表达式的相关函数 311
9.5 其他 312
9.5.1 foreach和迭代器（crowbar） 312
9.5.2 switch case（Diksam） 314
9.5.3 enum（Diksam） 315
9.5.4 delegate（Diksam） 316
9.5.5 final、const（Diksam） 319

附录A crowbar语言的设计 322
附录B Diksam语言的设计 336
附录C Diksam Virtual Machine 指令集 359

编程语言实用化指南——写在最后 369
参考文献 375





第 1 章

引子

1.1 为什么要制作编程语言

本书的主题是自制编程语言。单说现在被广泛使用的编程语言，就有 C、C++、Java、C#、Perl、Python、Ruby、PHP、Lisp、JavaScript 等。可能有人会质疑，既然已经有这么多语言了，真的有必要再特意创造一门新的语言吗？

实际上，自制编程语言还是大有益处的。

1. 可以帮助理解编程语言的内部运行机制

编程语言是程序员每天都要使用的工具。深刻地理解这个工具，对程序员来说非常重要。

一般来说，重新编写一个与已有程序相似的程序会被说成是“重复发明轮子”，这在行业内是不被认同的。但本书中想要实现的，偏偏是在众多语言存在的前提下再制作一门新的语言，正是“重复发明轮子”。这是深刻理解编程语言的最佳途径（缺点是要花很多时间）。

2. 能制作领域专用语言

比如在 Unix 的世界中，有 sed 和 awk 两种历史悠久的专为文本处理定制的语言（后来在此方向上发展出了 Perl 语言）。PHP 则是专门面向 Web 程序开发的语言。如果掌握了制作编程语言的技术，就可以在必要的情况下制作出领域专用语言（DSL，Domain-Specific Language）。

领域专用语言不一定会像 Perl 与 PHP 那么复杂，在很多情况下，如果能书写条件分支或者简单语句的话会方便许多，这也可以看作是一种专用领域。

比如在业务流程处理等软件中，很多时候为了切换测试环境与生产环境的数据库，需要重写配置文件，而这一操作经常会引发问题（比如由于版本升级需要增加配置文件项目，此时必须与旧版本配置文件合并）。这时候我们可能就会想，如果能直接在配置文件中写 `if` 语句将其按域名分开就好了。

除此以外，我们在填写数据时可能希望能支持类似 Excel 的简单算术公式，在玩游戏时希望能把游戏中的对话导出到一个外部文件中，等等。这些都可以看作专用领域并制作对应的 DSL。

3. 可以用编程语言扩展应用程序

将以上两方面的考量进一步延伸，我们就会得到以通用语言扩展某个应用

程序的构想。Emacs 这个编辑器就内置了 Emacs Lisp 这种 Lisp 方言，从而为 Emacs 的自定义提供了无限的可能性。同理，Microsoft Office 也可以使用 VBA 进行扩展。

对于这类应用程序扩展语言，当然完全可以使用某种已有的编程语言（Lua 等就在向这个方向发展），也可以在编写应用程序时从底层到扩展全部自己实现。这样就无需担心使用其他编程语言在版本升级时引起的兼容性问题了。

4. 说不定还会变成名人

如果自制的编程语言能在世界范围内得到广泛使用，那就太棒了。比如 Ruby 之父松本行弘先生就是世界名人。

不过坦白讲，通过自制编程语言来获得成功实在是太难了。即便语言被创造出来，如果没人用的话就不会产生相应的软件，这样就更不会有人用了。况且，即便真的因为发明了新的语言而变成了名人，通过这个赚到钱的希望也十分渺茫啊。其实我自己最近写的语法处理器都是免费发布的（不这样的话，语言没法普及呀）。

5. 自制编程语言非常有趣

啰嗦了这么多，说到底其实是因为自制编程语言非常有趣。

自制一门编程语言确实是一件非常有意思的事。有人说过“想写出终极程序的程序员，最终都去写操作系统或者编程语言了”，你可以通过自制编程语言感受到接触最核心技术的乐趣。

让尽可能多的人感受到这种乐趣，这正是本书的目标。

1.2 自制编程语言并不是很难

一提起自制编程语言，很多人都会觉得这是一件非常难的事情。

比如，即便是一个很常见的赋值语句：

```
a1 = b1 + b2 * 0.5;
```

在自制编程语言时都必须考虑到以下几个要点。

1. 需要将 a1、b1、b2 作为变量名解析出来。如果按照 C 语言的语法规则，变量名只能由字母或下划线开头，从变量名第二个字符开始才允许出现字母或数字。所以首先必须扫描这个语句，然后将匹配上述语法规则的部分提取出来。
2. 0.5 是一个含有小数点的常量，在提取这类常量时，能否用“数字组合 + 小数点 + 数字组合”来概括所有常量的特征呢（还要考虑是否允许 00.10 这样的数值）。当然我们的提取规则还要能处理 2 这样不含小数点的数值。
3. 乘法运算符 * 比 + 拥有更高的运算优先级，语句必须被解析为 b1 + (b2 * 0.5)。
4. b2 * 0.5 的计算结果，必须在与 b1 进行加法运算前就应该取得。也就是说对于复杂的计算，需要保存很多类似这样的临时运算结果。

假如你已经有了一定的编程经验，肯定能想到上面这些难点，甚至可以说你的编程经验越丰富，就越能感受到这其中隐藏着极大的难题。

不过，编程语言的语法处理器在 FORTRAN 诞生后已经经过了多年的研究，上面的这些难点都已经可以从前人那里找到解决方法 *。

* 当然，在早年原始的研发条件下，人们为了开发第一个编程语言编译器还是花费了相当大的精力，据说实现初版的 FORTRAN 编译器所花费的工时，累计达到了 216 人月 [1]。

在本书中，上面 1 ～ 3 的问题会用到名为 yacc 及 lex 的工具。问题 1 和问题 2 用 lex，问题 3 通过 yacc 解决。yacc 和 lex 都是非常老的工具了，现在流行的 LL 语言大多内置了 yacc。可能有人会说：“既然是以学习为目的去制作一门编程语言，如果还使用工具的话就太投机取巧了吧。”（这话很有道理。）所以在本书中，也会稍微介绍一下不使用这些工具的解决方法。

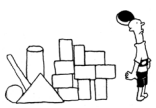
无论是使用工具，还是基于一些已有的解决方案自己编写，如果能掌握一些窍门的话，自制编程语言其实并不难。

那么你想不想试试自己制作一门编程语言呢？自己创造编程语言这件事情，不管怎么说都是很酷的吧。

1.3 本书的构成与面向读者

本书由以下的章节构成：

- 第 1 章　引子
- 第 2 章　试做一个计算器
- 第 3 ~ 4 章　制作无类型语言 crowbar
- 第 5 章　中文支持与 Unicode



- 第 6 ~ 8 章　制作静态类型的语言 Diksam
- 第 9 章　应用篇

第 1 章即是你正在阅读的章节。本章会对全书的构成以及讲解方式进行说明。

第 2 章通过制作一个简单的计算器，介绍 yacc/lex 的基本使用方法。其实讲解 yacc/lex 的部分，选择“计算器”为例实在有点老套，但确实没有比这更合适的题目了。此外还会介绍如何不依赖 yacc，使用递归下降分析器（Recursive Descent Parser）来制作一个计算器。

从第 3 章开始，会实际制作有一定行数规模的编程语言。

3 ～ 4 章会制作一个名为 crowbar 的无类型解释型语言，6 ～ 8 章则主要制作名为 Diksam 的支持静态类型的编译型语言（名字的由来会在后文提到）。在第 5 章中，会针对使用编程语言时的中文支持与 Unicode 问题进行说明。

第 9 章阐释闭包（Closure）及异常处理机制等进阶功能。

本书中会使用 C 语言作为编程语言语法处理器（编译器、解释器等）来编写语言（理由见后文中的具体说明）。而 crowbar 与 Diksam 最终都会累积为具备一定行数规模的程序（crowbar 约 8000 行，Diksam 约 2 万行）。

因此，阅读本书的读者最好具备两个条件：

1. 已经会 C 语言
2. 具备阅读较长代码的能力

不过无论哪个条件都不是必须的。

对于**条件 1**需要说一点的是，Java、C++、C# 等都是从 C 语言发展出来的语言，所以对于已经学习过这些语言的人来说，读 C 语言代码不会特别吃力。像预处理程序、指针等 C 语言特有的知识，建议你借此机会一并学习一下。因为至少就现阶段来说，无论是专家还是业余爱好者，但凡是程序员都免不了要用到 C 语言。而在 crowbar 或 Diksam 中，并没有使用很多 C 语言特有的功能。比如说不会出现 `*p++` 这种不易理解的写法，更多是写成数组下标的形式。

对于**条件 2**要说的是，虽然一个语法处理器整体来看是个上规模的程序，但是其基础构成的部分并不会很庞大。本书不会对每一行代码逐一进行注释，而是侧重于介绍解决问题的思路，所以如果仅仅是想阅读一下本书的话，是不需要具备阅读较长代码的经验的。但若你最后不满足于书中的讲解，还想要自己去阅读一下 crowbar 或者 Diksam 源代码的话，因为代码行数很多，编程经验尚浅的朋友

读起来可能会有压力。不过无论是业界还是外界人士，作为程序员总有一天会接触到大规模代码的程序，将本次实践作为入门的第一步也不是一件坏事。

综上所述：

如果你觉得自己不是本书所面向的读者，想办法加入其中不就行了？

所以无需担心什么，门槛其实没有你想的那么高。凡是对语法处理器有兴趣的朋友都是本书面向的读者。

1.4 用什么语言来制作

如前文所述，本书将使用 C 语言作为语法处理器的编写语言。

都什么年代了还用 C 语言？可能会有人这样想吧。其实就连我自己也会这样想。

但本书还是使用了 C 语言，其中一个理由是因为 yacc/lex 都是面向 C 语言的工具。

yacc/lex 本身是很老的工具。老工具虽然都有一些历史遗留问题，但也有其优点，即正是因为历史悠久，所以会积累下更详尽的技术文档。如前文所述，目前的 LL 语言大多使用 yacc。

另一个使用 C 语言的理由是：想要降低“依赖程度”的话，C 语言是最适合的。

比如说用 Java 编写软件，运行环境中必须安装 JVM（Java 虚拟机）。如果用 C# 则必须要安装 .NET Framework。在自制编程语言的理由中，我们曾经列举了“可以用编程语言扩展应用程序”这一条，并且提到，如果能在编写应用程序的时候从底层到扩展全部自己实现会更加放心，其目的就是为了不依赖 JVM 或 .NET Framework。这样在 Java 或 .NET 版本升级时也就无需操心了。

此外考虑到组合各种应用程序这个用途，C 语言在众多编程语言中可以说是最具通用性的。无论被组合的应用程序采用何种语言编写，毫无疑问都可以调用 C 语言。

1.5 要制作怎样的语言

1.5.1 要设计怎样的语法

编程语言有很多种，C、C++、Java、C# 等都是**面向过程的编程语言**（C++、Java、C# 虽然也被称为面向对象，但可以把面向对象看作是面向过程的一个派生）。目前看来，虽然面向过程的语言是主流，但还存在 Haskell、ML 这样的**函数式编程语言**。函数式编程语言就是“变量值无法被更改”的一种语言 *。

* 从这个定义来说，Lisp 严格讲还不能算是函数式编程语言。

对于已经习惯了面向过程语言的人来说，肯定会想“变量值无法更改还怎么写程序呀”。其实这类语言已经编写出了很多实用的程序。在函数式编程的基础上发展出了如 Prolog 这样的**逻辑编程语言**以及被称为**并行程序设计语言**的 Erlang。

不过目前被广泛使用的仍然是面向过程的编程语言，本书中的代码示例使用的也都是面向过程的语言风格，当然里面还会加入面向对象的一些功能实现。在本书中，除了会有 C++、Java、C# 这种基于类的面向对象之外，也会涵盖类似 JavaScript 这种没有类的面向对象。

语法层面上，会使用类似 C 语言的风格。crowbar 的示例代码如代码清单 1-1 所示，Diksam 的示例代码如代码清单 1-2 所示。

代码清单 1-1
crowbar 版 FizzBuzz

```
for (i = 1; i < = 100; i++) {
    if (i % 15 == 0) {
        print("FizzBuzz\n");
    } elsif (i % 3 == 0) {
        print("Fizz\n");
    } elsif (i % 5 == 0) {
        print("Buzz\n");
    } else {
        print("" + i + "\n");
    }
}
```

代码清单 1-2
Diksam 版 FizzBuzz

```
int i;

for (i = 1; i < = 100; i++) {
```

```
    if (i % 15 == 0) {
        println("FizzBuzz");
    } elsif (i % 3 == 0) {
        println("Fizz");
    } elsif (i % 5 == 0) {
        println("Buzz");
    } else {
        println("" + i);
    }
}
```

顺便说一下这个名为 FizzBuzz 的小程序，其运行机制如下：

> 输出从1到100的数字，如果为3的倍数时，则将数字替换为Fizz，5的倍数时则输出Buzz，同时为3与5的倍数时输出FizzBuzz。

这个小程序引自下面的文章。文章大意是建议企业在面试程序员时，至少应聘者能写出这种程度的代码再考虑录用。

> ◎为什么自称程序员的人写不出程序?
> http://www.aoky.net/articles/jeff_atwood/why_cant_programmers_program.htm

看了示例就能明白，无论 crowbar 还是 Diksam，都是与 C 语言非常类似的语言。

如上所述，本书虽然会创造一门新语言但仍然会用到 C 语言，所以本书所面向的读者应该是已经掌握了 C 语言的（还没有掌握的人可以先去学习一下）。因此如果选择 C 语言风格的语法，读者应该会感到很亲切，更重要的是笔者本人已经习惯了 Java、C# 这种以 C 语言为基础的编程语言。

C 语言是很老的语言了，这门语言不是在前期经过严谨的设计，而是在项目中一边实践一边慢慢发展起来的，因此语法上难免有很多考虑不周的地方。比如在 C 语言中赋值使用 =，即数学中的等号。而 C 程序员在初学者阶段编写 if 语句时，肯定免不了会写成这样：

```
if (a = 0) {    ←应该写 "==" 但是写成了 "="
    ⋮
}
```

这样惨痛的教训至少也要经历一次吧。赋值在 Pascal 等语言中，一般使用 :=。如果让一个没有编程经验的人来学习，Pascal 这种语法应该更加友好一些。

不过我现在是要制作一门新的编程语言，而使用这门新语言的人应该都已经习惯了 C 语言的运算符，如果这里将赋值运算符定为 := 的话反而会引起混乱，

说不定我自己就先头晕了。所以经验之谈是，语法上的些许优劣还是要给“习惯”让步的。

——出于这种考虑，我最终决定制作一门与 C 语言类似的编程语言。

决定语法风格是编程语言创造者的特权。如果顾虑用户习惯，可以参考并整合已有的编程语言。当然，也可以完全不考虑用户的感受，去创造一门“理想的语言”。虽然我是以 C 语言的语法为基础，但还是想到了以下几点可以改进的地方。

1. if条件在C语言中，如果按条件执行的语句只有一句，则{}可以省略。但是这经常会造成混乱，很多项目的编码规范中都会规定必须包含{}。因此最好在语法层面直接将{}设置为不可省略（crowbar、Diksam均如此）。
2. 既然已经将if条件中的{}设置为不可省略，那么if后面的()要怎么办呢？（关于这一点，我起初在crowbar中尝试了一下省略if的括号，结果发现在crowbar中()是不可省略的。）
3. 伴随着语言的逐步完善，考虑到要增加一些关键字（参考2.3.1节的补充知识），此时再处理与已存在程序的变量名相冲突的问题就比较麻烦，所以考虑在所有的变量前加上$（Perl或PHP等的解决方式），或者将关键字全部以大写字母开头（Modula-2等的解决方式）。
4. switch case语句中，最好能去掉忘了写break就会进入下一个case这种容易产生问题的设计（Java没有改进这一点，C#则做了一些半吊子的改进）。
5. switch case语句中，如果没有进入任何一个case条件分支，也没有写default分支，那么在运行时直接报错会不会更好一些（Pascal就是这样处理的）？
6. 编码规范通过缩进来约束怎么样？比如像Python那样通过缩进来表明逻辑结构。
7. 对于我来说，阅读Python风格的代码还有些吃力，因此是不是做成像C语言那样用花括号包裹语法块、把强制缩进的检查交给编译器去做比较好呢？

我希望读者朋友们也能够用好语言开发者的特权，不断去追求“更加理想的语言”。呃，虽然我这样讲可能会被说成是站着说话不腰疼吧。

1.5.2 要设计怎样的运行方式

程序员中应该无人不知，编程语言有编译型语言和解释型语言两种。

编译型语言中，C 和 C++ 比较有代表性。这类语言通常会将程序员编写的程序源代码，最终输出为机器码的可执行文件。

但是想要输出机器码的话，必须首先掌握机器码才行。即便学习了机器码并

写出了编译器，该编译器也无法输出供其他型号 CPU 运行的文件 *。

* 为了解决这个问题，一般的编译器都会将依赖 CPU 生成的机器码的部分单独归为一个名为 Backend 的模块，根据不同的 CPU 可以更换相应的 Backend，就可以支持其他型号的 CPU 了。

这类生成机器码的编程语言的优点是运行速度非常快，但是编译器性能优化的相关技术，学习起来非常有难度。另外，在自制编程语言的理由中曾经列举了“可以用编程语言扩展应用程序”这一点，而输出机器码的编译器并不适合这个用途。因此本书中会选择解释型语言。

虽说“解释型语言”只是一个词，但是其实现方法又分很多种。

解释型语言的“解释”一词源自英语的interpreter，是“能进行翻译的物体”的意思。编译器将源代码翻译为机器码，之后 CPU 直接运行机器码就可以了。与此相对的解释型语言，则将程序员编写的源代码通过解释器这一程序一边解析一边运行——这种公式化的定义看起来只有简单的两个步骤，但现实中几乎不存在这么单纯的解释型语言（DOS 的批处理脚本或 UNIX 的 SHELL 脚本是最接近解释型语言的定义的）。虽说名为“解释型语言”，但其中的大多数都会将源代码临时转换为某种中间形态。

比如有代码清单 1-3 这样的代码。

代码清单 1-3
简单的 if 语句 *

```
if (a == 10) {
    printf("hoge\n");
} else {
    printf("piyo\n");
}
```

* 代码中的 hoge、piyo 这两个单词，经常在输出无意义的语句时使用（多见于日本，英语国家则较多使用 foo、bar）。详细请参考以下的页面：http://avnpc.com/pages/devlang#hoge

从机器的角度看，源代码其实只是一些文字的排列组合而已，机器是无法直接运行的。现在大多数编程语言，都会将代码转换成一种叫分析树（parse tree，也叫语法分析树或语法树）的东西。上面的代码如果做成分析树，则如图 1-1 所示。

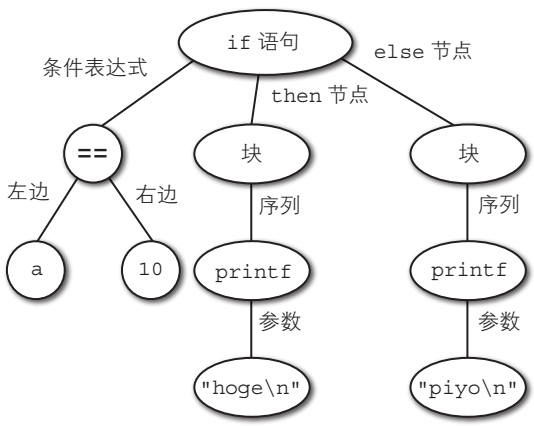
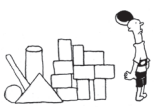


图 1-1
分析树示例



* Perl 6 还不知道什么时候出来，就不管它了 :-)

Perl*、Ruby 等语言，一旦将代码转换为分析树后，分析树将无法再还原回源代码。

本书第 2 章以后所用到的语言 crowbar 就是采用这种运行方式的语言。

对于这类语言来说，从源代码到分析树的构建过程还是得称为“编译”。但是这里的编译器是在程序启动时自动执行的。由于分析树会生成在内存里，因此不会生成目标代码或目标文件，所以程序员（用户）一般意识不到有编译器在执行。这类语言如果存在语法错误，会在刚开始运行时就被报出来，这正是源代码被一次性全部读入并构建分析树的证明。如果是纯粹的解释型语言，如批处理脚本或 SHELL 脚本，则会运行到有语法错误的地方才会报错。

那么，相对于 Perl、Ruby 这样的运行分析树型语言，在 Java 等语言中，取代分析树的则是更底层的字节码，然后通过解释器运行字节码。字节码只是一些简单的数字排列，为了尽可能地让人读懂字节码，字节码中的所有指令都被加上了一些名为助记符（mnemonic）的字符，代码清单 1-3 的源代码经过这样一番处理之后最终会变成代码清单 1-4 的样子（源代码中的 `printf` 改为 `System.out.println`，并使用 javap 输出）。

代码清单 1-4
Java 的字节码

```
 0: bipush 10
 2: istore_1
 3: iload_1
 4: bipush  10
 6: if_icmpne       20
 9: getstatic
12: ldc
14: invokevirtual
17: goto    28
20: getstatic
23: ldc
25: invokevirtual
```

本书第 5 章以后所用到的语言 Diksam，就是采用这种运行方式的语言。

在 Java 中，编译器生成的字节码会被保存在 class 文件中。但是在 Diksam 中，编译器会在程序启动时执行，因此字节码保存于内存中，不会生成类似 class 文件的东西。由此可以看出，从用户的角度出发，不需要意识到 Diksam 内部其实有字节码在执行。Python 也是使用了类似的处理机制。

补充知识 “用户”指的是谁?

前文曾写道“因此程序员（用户）一般意识不到有编译器在执行。”

通常来说，用户是指使用程序员编写的程序的人，但是在这里，因为我们是要制作一门编程语言，所以本书中的用户应该是指使用我们制作的编程语言的人，即程序员。

这种指代在操作系统、类库、编程语言等面向程序员的文档中经常出现，不过可能有读者会有误解，在此特别补充说明一下。

补充知识 解释器并不会进行翻译

在很多入门书中，提到编译器与解释器时，一般会采用以下说明：

> **编译器会将源代码一次性全部翻译为机器码。**
> **与此相对的解释器，不会事先做一次性翻译，而是在运行的同时，逐行分块地将源代码翻译为机器码。**

请允许我说句老实话，这样的说明是完全错误的。

解释器会将源码或分析树解析为字节码这种中间形态，并且一边解析一边运行，但是解释器并不会将源码翻译为机器码。

Java 或 .NET Framework 都具备在运行的同时将字节码转换为机器码的功能，这叫作“JIT（Just-In-Time）编译”技术，而这部分技术并不属于解释器。

那么解释器具体是如何运行程序的呢？读到后面你就会明白了。

1.6 环境搭建

1.6.1 搭建开发环境

本书的开发语言是 C 语言，辅助工具是 yacc 和 lex。

UNIX（包含 Linux 等）大部分都已经预装了开发所需的 yacc 和 lex，当然也有例外，而 Windows 则默认没有预装。不过无需担心这些，我们完全可以全部使用自由软件来搭建一个可用的开发环境。

那么，下面我们就开始介绍这些软件的获取途径。

1. C 编译器

免费的 C 编译器可以使用 GNU 项目提供的 GCC（GNU Compiler Collection）。

Linux 等免费的 UNIX 环境下大多都预装了 GCC*。Windows 下可以使用 MinGW（Minimalist GNU for Windows）。

* 最近 Linux 不预装 GCC 的情况似乎越来越多了。

可以从下面的 URL 下载。

```
http://www.mingw.org/download.shtml
```

安装 MinGW 时，UNIX 环境下的程序会将构建（build）时使用到的 make 工具也一并安装。不过，安装完毕后可执行文件名有点奇怪，是 `mingw32-make.exe`，我将其复制并重命名为 `gmake.exe` 以方便使用。

2. cygwin 或 MSYS

cygwin 是可以运行在 Windows 上的类 UNIX 环境。比如说想在命令行提示符中列出当前文件夹内的文件时，Windows（DOS）会使用 `DIR` 指令，UNIX 则使用 `ls` 指令。一般用惯了 UNIX 的人，往往会在 Windows 的命令行提示符中不自觉地敲出 `ls` 却尴尬地发现指令不存在，而安装了 cygwin 就可以避免这样的情况发生。那么对于不经常使用 UNIX 的人还有必要装 cygwin 吗？因为在后文中提到的 bison 要使用 UNIX 中的 m4 工具，所以无论是 cygwin 还是 MSYS，至少还是要安装其中一个的 *。MSYS 与 cygwin 都是在 Windows 上模拟 UNIX 环境的软件。

* m4 其实也可以单独安装，但似乎没有独立的安装包，可能会非常麻烦。

cygwin 可以从下面的网址中获取：

```
http://cygwin.com/
```

MSYS 可从 MinGW 页面中下载：

```
http://www.mingw.org/download.shtml
```

此外，因为 cygwin 也包含 GCC，可以没有 MinGW 而通过 cygwin 安装 GCC。但是使用 cygwin 安装的 GCC 编译，运行时需要依赖 cygwin1.dll 文件，在其他机器运行还需要把 DLL 也复制过去，所以还是使用 MinGW 更方便。

3. bison

如果环境无法直接运行 yacc，可以使用 GNU 项目提供的 bison。

```
http://gnuwin32.sourceforge.net/packages/bison.htm
```

4. flex

同理，如果环境无法直接运行 lex，可以使用 lex 的免费版 flex。

```
http://gnuwin32.sourceforge.net/packages/flex.htm
```

补充知识　关于 bison 与 flex 的安装

bison 由 GNU 项目提供。GNU 项目是由理查德·斯托曼（Richard Matthew Stallman）创立的项目，目标在于建立一个完全相容于 UNIX 的自由软件环境。

GNU 项目提供的软件的许可证为 GPL（通用公共许可协议，General Public License）。粗略地说，GPL 是这样一种许可证：

- **发行 GPL 的程序时，必须公开源代码并且声明源代码的出处；**
- **包含 GPL 源代码的程序，必须受 GPL 许可证条款约束；**
- **程序即使以动态链接方式使用 GPL 程序，也必须受 GPL 许可证条款约束。不过这个限制在 LGPL 许可证（Lesser GPL）中有所放宽。**

也就是说，你的程序中只要用到 GPL 的程序，哪怕这部分再小，你的程序也会自动变成 GPL 程序，必须与源代码同时公开。这对于那些为了防止盗版而不得不采取一些措施的商用软件来说简直是致命的。因此也有人戏称 GPL 的这个特性是“GPL 传染”或“GPL 病毒”。

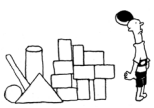
那么 bison 是否也是如此呢？后文会有说明，bison 的作用是将用户编写的配置文件输出为 C 语言格式的代码。这里的 C 代码中会包含一些属于 bison 的代码。那么是不是说使用 bison 去制作编程语言，所做出的编程语言在发行上也必须遵守 GPL 许可证呢？关于 bison 输出的 C 代码这一点，是 GPL 的一个特例，可以不受 GPL 许可证约束。此处在 GNU 项目有关 GPL 的 FAQ 页面中有如下的记载：

> **碰巧的是，Bison 也可以用于开发非自由软件。这是因为我们明确允许在 Bison 的输出结果中包含的 Bison 的标准解析程序可以不受限制。我们做此决定，是因为已经存在与 Bison 类似的工具被用于非自由软件的开发。**
> http://www.gnu.org/licenses/gpl–faq.ja.html

另一方面，flex 则是遵循 BSD 许可证（Berkeley Software Distribution，加州大学伯克利分校开发的软件套件集合）的（不是修订版 BSD）。BSD 许可证的程序再次发行时，文档中必须要附加 BSD 的版权信息。

flex 会像 bison 一样输出 C 代码，这里的 C 代码也像 bison 一样，会包含一些属于 flex 的代码。但是这部分代码并不需要附加 BSD 的版权信息。因为 flex-2.5.34 携带的 COPYING 文件中有这样的描述：

Note that the “flex.skl” scanner skeleton carries no copyright notice. You are free to do whatever you please with scanners generated using flex; for them, you are not even bound by the above copyright.

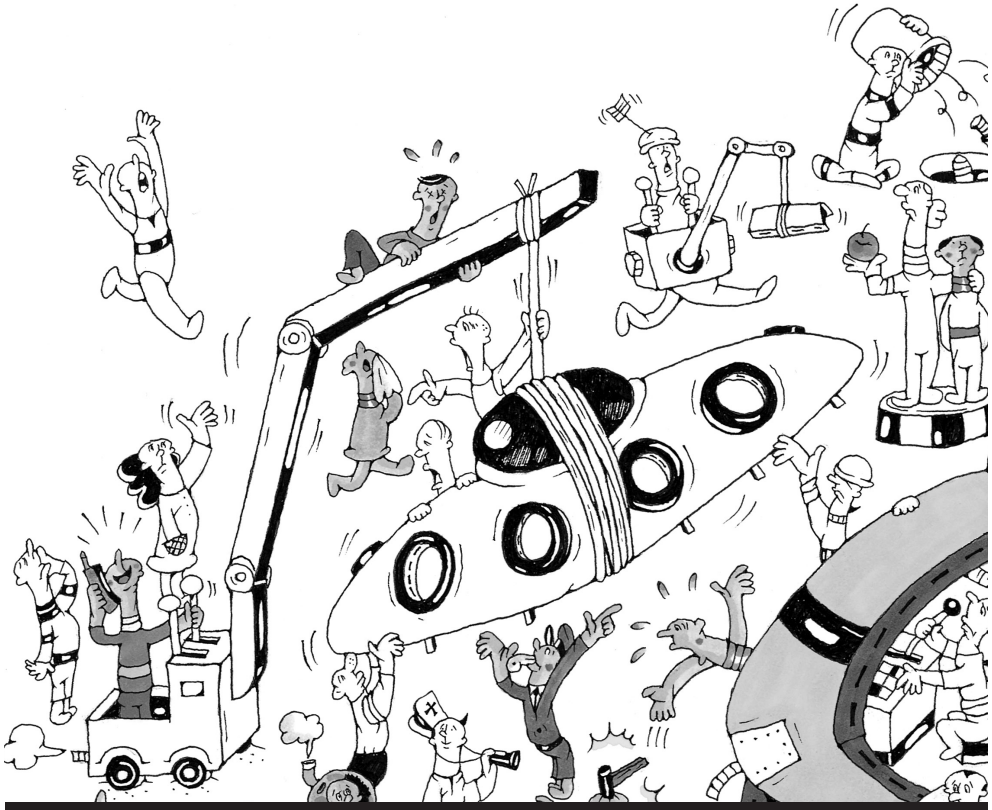


1.6.2 本书涉及的源代码以及编译器

本书所涉及的源代码，可以在作者的网站上下载：

```
http://avnpc.com/pages/devlang#download
```

在开始撰写本书之前，crowbar 和 Diksam 就已经存在一些公开的版本了，本书所用到的代码都对其进行了重新的整理和修正，因此本书相关的代码将重新以 book_ver 作为版本号。比如本书最开始制作的 crowbar 的版本号就是 crowbar book_ver.0.1。



第 3 章

制作无类型语言 crowbar

3.1 制作 crowbar ver.0.1 语言的基础部分

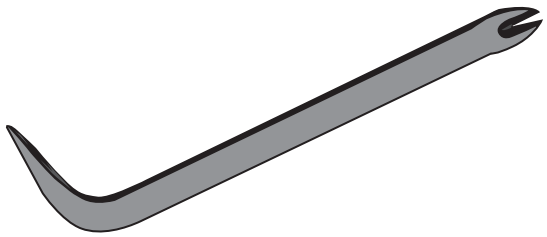
本书首先制作一门无变量类型的语言。像 Perl、Ruby、Python、PHP 这些近些年火起来的脚本语言，基本都没有变量类型。我们把将要制作的语言命名为 crowbar。

本章首先对 crowbar 的初始版本（ver.0.1）进行简要说明。

3.1.1 crowbar 是什么

crowbar 不是那种如果找到有四片叶子就会有好运降临的植物（那叫三叶草），而是如图 3-1 这样形状的工具。

图 3-1
名为 crowbar 的工具



之所以起名叫 crowbar，主要是因为这次要做的语言会生成分析树并执行。单就这点来说是与 Perl 比较接近的。有句话是怎么说来着，对了，就是那句经常能从新闻里听到的：

撬棍状的物体 [①]

于是我就以 crowbar 命名了。喂，别向我扔石头啊。

如前文所述，crowbar 的语法应当照顾本书读者的习惯，所以沿袭了 C 语言的语法。

首先将初版的 crowbar 命名为 crowbar book_ver.0.1，示例代码如代码清单 3-1 所示。

① 当刑事案件发生时，如果物证尚不充分，警方在新闻发表会上描述犯罪所使用的道具时会经常用“撬棍状的物体”来形容，这一说法对于日本人来说是耳熟能详的。由于 Perl 与撬棍在日语中发音很接近，所以作者用 crowbar 命名其实是一语双关的小幽默。——译者注

代码清单 3-1
fizzbuzz_0_1.crb

```
1: for (i = 1; i < = 100; i = i + 1) {
2:     if (i % 15 == 0) {
3:         print("FizzBuzz\n");
4:     } elsif (i % 3 == 0) {
5:         print("Fizz\n");
6:     } elsif (i % 5 == 0) {
7:         print("Buzz\n");
8:     } else {
9:         print("" + i + "\n");
10:     }
11: }
```

与代码清单 1-1 不同的是，由于自增运算符 ++ 尚未实现，所以写成了 `i = i + 1`。

这个版本的 crowbar 还没有实现一门编程语言应当具备的所有基本功能（可能有读者会说，就这样也敢与 Perl 相提并论呀），当前版本所实现的功能，会在以后的章节中加以说明。

3.1.2 程序的结构

crowbar 与 Perl 一样，支持在**顶层结构**书写代码。所谓的顶部结构，即函数或类的外侧。

C 语言中，在函数的外面可以定义变量却不能书写执行语句，因此即便只写一句“hello, world”，也需要 `main()` 函数。Java 就更悲惨了，必须写长长的一串 `public class HelloWorld` 还有 `public static void main(String[] args)` 这种外行人看来像咒语一样的东西。如果仅仅想写几行简单的脚本，这实在很麻烦，而对于初学者来说也增加了学习的难度。

在 crowbar 中，如果想写一个显示“hello, world”的程序，只需简单地写成下面这样就可以了。

```
print("hello, world\n");
```

无需再包裹函数或者类。

函数的定义，需要使用保留字 `function`，按如下方式书写：

```
# 显示将 a 与 b 相加的值，并且作为返回值返回的函数
function hoge(a, b) {
    c = a + b;
```

```
    print("a+b.." + c + "\n");

    return c;
}
```

函数定义在程序中可以写在任意位置。程序执行时，首先将顶层结构中的语句从上往下顺序执行，函数定义部分会被跳过。直至函数被调用时，才执行该函数内的语句。

函数如果不存在 return 语句，将返回特殊的常量 null。

3.1.3 数据类型

可以使用的数据类型如下所示。

- 布尔型。值可以为 true 或 false。
- 整数型。其实就是 crowbar 底层运行环境的 C 语言的 int 型。
- 实数型。即 crowbar 底层运行环境的 C 语言的 double 型。当整数型与实数型混合运算时，整数型将被扩充为实数型。
- 字符串型。可以通过 + 运算符连接。另外，如果字符串在左侧数值在右侧，用 + 连接的话，右侧将被转换为字符串型。

例如：

```
print("10 + 5.." + (10 + 5));      ←将显示 10 + 5..15
```

- 原生指针型（Native Pointer）。请读者不要根据名字将其想象成那种可以直接访问内存的邪恶指针，crowbar 的原生指针型类似于 C 语言的 FILE*，是用于在 crowbar 内部移动跳转的类型。详细请参考 3.1.7 节。

在 book_ver.0.1 中，不存在数组、关联数组（associative array）、类、对象等类型。

3.1.4 变量

crowbar 与 Perl、Ruby 等相同，都是静态无类型（即变量无需声明类型）语言。

crowbar 无需变量声明，赋初始值时就包含了声明过程（和 Ruby 非常类似）。

如果直接引用一个还没有赋值的变量则会报错。

变量的命名规则与 C 基本一样，必须以字母开头，第二个字符开始可以使用字母数字，也支持下划线。与 Perl 等不同的是，变量开头无需书写 $ 符号。

函数内首次进行赋值的变量会作为函数的局部变量，局部变量的生命周期及作用域仅限于当前函数内部。C 语言等还可以在函数中用 { } 再开辟一个块（Block），并在块内有更小作用域的局部变量，crowbar 则不支持这种特性。

变量是在赋值语句执行时进行声明的，如下例所示：

```
if(a == 10) {
    b = 10;
}
print("b.." + b);
```

a 只有为 10 的时候 b 才被声明，print 语句可以正常显示。如果 a 不为 10 则会报出未定义变量的错误。

在顶层结构中赋值的变量会成为全局变量。函数中引用全局变量时，需要用 global 语句进行声明。

global 语句可以按以下的方式使用：

```
global 变量名 , 变量名 , …;
```

比如函数内用 global a; 声明之后，在该函数内就可以引用全局变量 a（如果全局变量 a 不存在则会报运行错误）。

比如运行代码清单 3-2，运行结果如下所示：

```
a..30
a..20
```

运行结果第 1 行的 a..30 是代码清单 3-2 第 10 行的 print 输出结果，因此这里显示的是 func2() 中被赋值的全局变量 a 的值。

第 2 行的 a..20 则是第 15 行的 print 结果，显示的是局部变量 a 的值。

因为有了 global 语句，所以第 5 行赋值的是全局变量 a 的引用，而第 9 行只引用了局部变量，因此即使对其赋值也不会对全局变量产生影响。

代码清单 3-2
global.crb

```
1: a = 10;          ←定义全局变量 a 的声明
2:
3: function func() {
4:     global a;
5:     a = 20;      ←这里的 a 是全局变量
6: }
```

```
 7:
 8: function func2() {
 9:     a = 30;   ←这里的a是局部变量
10:     print("a.." + a + "\n");
11: }
12:
13: func();
14: func2();
15: print("a.." + a + "\n");
```

那么，为什么一定要使用global语句声明后才可以引用全局变量呢？这样的设计有以下两个原因。

- 如果没有任何约束就可以直接引用全局变量，那么编写函数时就必须随时掌握所有全局变量的情况，而对于强调高内聚性的函数来说，这种设计会产生致命的错误。
- 全局变量的使用频率并不高，因此设置这样一点障碍对编写程序不会产生太大影响。

话虽如此，在使用STDIN（标准输入的文件指针）这样的全局变量时也必须声明，还是多少有些不方便的。

补充知识　初次赋值兼做变量声明的理由

如上文所述，crowbar会在变量初次赋值时兼做变量声明，即如果直接使用没有赋值的变量会报错。

比如在Perl中，默认情况下，即使没有赋值的变量仍然可以使用。此时该变量值会根据上下文自动转换。像下面这样书写的话：

```
print 123 * $a; #对未赋值的变量$a进行乘法运算
```

运行结果为0，因为未赋值的变量$a的值被自动转换为0了。

但是这样的设计容易因为变量名输入有误而引起BUG*。因此在crowbar的设计中，只能使用进行过初次赋值的变量。

* 上面的语句在Perl中如果加上-w参数并运行的话会出现警告。

还需要注意的是，crowbar在执行变量的赋值语句时才会被声明，而Ruby只要书写了赋值语句就完成了变量声明，即赋值语句的执行不是必须的。因此，像下面这样：

```
x = x; #这个例子中，赋值语句执行前，x也可以使用
```

或

```
if false
  a = 1
end

print a; #赋值语句没有执行，也可以使用a。
```

这些程序在Ruby中都是合法的。关于这样设计的理由，Ruby的作者松本行弘先生做了如下说明（请参考ruby-list邮件列表的No.33798）：

全局变量的作用域应当通过静态方式决定，也就是说，在赋值语句开始执行才检查变量是否存在，这样的设计并不好。
因为动态的变量作用域用户理解起来有难度，同时也失去了一次编程语言中为数不多的可以进行性能优化的机会。

关于这一点我是持同意态度的 *，那为什么 crowbar 中没有这样去做呢？理由其实很简单，只是想要偷懒一下而已。

* 不过我还是有些介意，Ruby 中连类或方法的定义都是动态可执行的，为什么偏偏变量的定义要做成静态的呢。

补充说明　各种语言的全局变量处理

下面来看一看其他语言中全局变量的处理方法。

Perl：变量默认是全局的，只有加上 `local` 或 `my` 等定义后才会变成局部变量。
Ruby：用 `$` 开头的变量是全局变量。
PHP：与 crowbar 一样，函数内要引用全局变量的话，用 `global` 语句定义。

一般来说，程序中应该避免到处使用全局变量，而尽可能优先保证局部变量的内聚性。从这个角度来讲，Perl 式的设计是不能借鉴的（当然如果是一次性的脚本，这样倒是很方便）。Ruby 式的设计是比较合理的，但按这个设计写出来的程序可能到处是记号，丧失了程序的美感（这只是我主观的感受）。因此 crowbar 采用了 PHP 风格的 `global` 语句的设计。

3.1.5 语句与结构控制

crowbar 与 C 语言一样，有 `if`、`while`、`for` 等结构控制语句。

与 C、C++、Java 等语言有以下两处比较大的区别：

- crowbar 中不允许出现悬空 else（花括号 {} 是强制书写的）；
- 因为不允许悬空 else，所以引入了 `elsif` 语句。

具体来说是下面这样的形式：

```
# if 语句的例子
if (a == 10) {
    # a == 10 时执行
} elsif (a == 11) {
    # a == 11 时执行
} else {
    # a 不为 10 也不为 11 时执行
}
```

```
# while 语句的例子
```

```
while (i < 10) {
    # i 比 10 小时，此处循环执行
}
```

```
# for 语句的例子
for(i = 0; i < 10; i = i + 1) {
    # 这里循环 10 次
}
```

此外，在 crowbar 中也可使用下列语句，其意义与 C 语言相同。

- break：从最内层的循环中跳出。
- continue：跳过最内层循环中剩余的代码。
- return：从函数退出，并将后面的值作为返回值返回。

break 或 continue，最好能像 Java 那样附加一个标签，但当前版本还没有这个功能（book_ver.0.3 实现了标签功能）。

补充知识　elif、elsif、elseif 的选择

C 等语言中，if 语句允许没有花括号的写法（也称作悬空语句），也可以像下例这样用 else if 排列书写。

```
if (a < 10) {
    ⋮
} else if (a < 20) {
    ⋮
} else if (a < 30) {
    ⋮
} else {
    ⋮
}
```

C 或 Java 虽然设置了这种特别的结构控制语法，但偶尔也有初学者会误解其意义，以为 else if 不是一个专用语句，而是 else 语句后省略花括号又写的一个 if 语句。说起来在工作中的确会遇到很多项目，在编码规范中明确规定了“禁止省略花括号”，这样就可以放心地去写 else if 了。

crowbar 中直接废弃了悬空语句，无法书写上述形式的 else if，为此特别引入了 elsif。不过不同语言对于 elsif 的设计都不太一样，实在让人有些头疼。

B Shell、Python、C 预处理器	elif
Perl、Ruby、MODULA-2、Ada、Eiffel	elsif
Visual Basic、PHP	elseif

因为 crowbar 的目标就是成为“Perl 那样的东西”，所以我就私自决定采用 elsif 了。

3.1.6 语句与运算符

首先，crowbar 支持以下形式的常量作为语句。

- 整数字面常量，如 123 等。
- 实数字面常量，如 123.456 等。
- 字符串字面常量。双引号包裹的字符串，如 "abc" 等。

另外变量也可以作为语句。

进而可以和运算符结合构成更复杂的语句，当然还支持括号。

crowbar 可使用的运算符如表 3-1 所示（按运算优先级排序）。

表 3-1 crowbar 可使用的运算符

–（单目取负）	符号的反转
* / %	乘法、除法、求余
+ –	加法、减法
> >= < <=	大小比较
== !=	同值比较
&&	逻辑与
||	逻辑或
=	赋值

% 运算也可以用在实数上，本质上是在内部调用了 C 的函数 fmod()。

无论 C 语言还是 crowbar，都没有用常量直接表示负数。想使用负数时，可以使用单目取负符 -。

而与 C 语言一样，&&、|| 都是短路运算符。也就是说，像下面这样的条件语句：

```
if (a < 10 && b < 20) {
    ⋮
```

当 a < 10 的条件不成立时，不再判断 b < 20 这一条件语句（已经短路，所以表达式无论真伪都不会在 if 语句中执行）。

3.1.7 内置函数

内置函数是 crowbar 最开始就包含的用 C 语言编写的函数。crowbar 当前版本的内置函数如表 3-2 所示。

表 3-2
crowbar book_ver.0.1 的内置函数

函数名	功能
print(arg)	显示 arg。arg 的类型可以是整数、实数、字符串
fopen(filename, mode)	打开一个文件，返回文件指针。mode 的可选参数与 C 语言的 fopen() 一样（其实就是原封不动的传给了 C 语言）
fclose(fp)	传入 fp 即关闭文件
fgets(fp)	从 fp 中读出一行字符串并返回
fputs(str, fp)	向 fp 输出字符串，输出时不会自动添加换行

显而易见，基本上所有文件操作函数的设计都沿袭了 C 语言的 stdio.h。只是因为 crowbar 有字符串类型，所以 `fgets()` 等的用法会稍有不同。

此外，`fopen()` 返回的类型是 crowbar 才有的“原生指针型”。上例中只是单纯指向 C 的 `FILE*`，但是这个类型的特殊之处远不止于此。比如用内置函数实现 GUI 时，创建一个打开新窗口的函数 `create_window()`，其返回值应当能表示一个“窗口”，此时就可以考虑使用原生指针型来实现。

crowbar 中已经默认声明了 `STDIN`、`STDOUT`、`STDERR` 等全局变量，分别对应 C 语言中的 `stdin`、`stdout`、`stderr`。

3.1.8 让 crowbar 支持 C 语言调用

考虑到 crowbar 的用途之一是扩展应用程序，那么应当让 C 语言编写的其他应用程序可以很容易地调用 crowbar 解释器。

代码清单 3-3 是与当前版本 crowbar 所属的 main.c 基本一样的代码段。调用里面这些函数，需要用 `#include` 包含 CRB.h 文件。

代码清单 3-3
crowbar 被 C 语言调用

```
CRB_Interpreter     *interpreter;
FILE *fp;
/* 中间省略 */

/* 生成 crowbar 解释器 */
interpreter = CRB_create_interpreter();

/* 将 FILE* 作为参数传递并生成分析树 */
CRB_compile(interpreter, fp);

/* 运行 */
CRB_interpret(interpreter);
```

```
/* 运行完毕后回收解释器 */
CRB_dispose_interpreter(interpreter);
```

3.1.9 从 crowbar 中调用 C 语言（内置函数的编写）

反过来，从 crowbar 中调用 C 语言的函数（内置函数）也同样容易。

首先用 #include 包含面向开发人员的头文件 CRB_dev.h，像下面这样表示 C 函数：

```
CRB_Value hoge_hoge_func(CRB_Interpreter *interpreter,
                         int arg_count, CRB_Value *args)
{
    /* 中间省略 */
    return value;
}
```

这里调用的 interpreter 是指向解释器的指针，arg_count 代表向该函数传递的参数的数量，args 是参数的值（CRB_Value 类型详见 3.3.8 节）。

crowbar 是无类型语言，因此参数的数量与类型的检查都必须在内置函数进行。

通过这种方式制作出的 C 函数，通过 CRB_add_native_function() 函数即可注册到解释器中，成为 crowbar 的内置函数。

```
/* 将 C 的函数 hoge_hoge_func 注册为一个 crowbar 可以调用的内部函数
   并命名为 hoge_hoge */
CRB_add_native_function(interpreter,
                        "hoge_hoge", hoge_hoge_func);
```

3.2 预先准备

crowbar 的语法处理器有一定的行数规模（最终版有 8000 行左右），因此应当预先约定编码规范，并准备好底层的库。

那么让我们暂时离开语法处理器，先准备下面这些事项吧。

3.2.1 模块与命名规则

crowbar 由以下 3 个模块构成：

- crowbar 主程序（CRB）
- 内存管理模块（MEM）
- Debug 模块（DBG）

括号中的 CRB、MEM 等是模块名。

这里我所指的模块，即可以完成某些特定功能的程序块。一个模块中基本都会包含多个 .c 文件。

MEM 与 DBG 均为通用模块，并不是 crowbar 专用的。代码分别位于 crowbar 文件夹下的 memory、debug 子文件夹中。

C 语言中没有 C++ 和 C# 的命名空间，也没有 Java 中的包机制，因此必须制定命名规范来避免可能出现的命名冲突。因此我们使用以下的命名规范。

1. 模块必须有前缀3个字母的缩写（如：CRB）。
2. 类型名，以大写字母开始，并使用大写字母连接单词（如：CRB_Interpreter）。
3. 变量名/函数名，全部使用小写字母，使用下划线连接单词（如：alloc_expression()）。
4. 宏命名为全大写字母，使用下划线连接单词（如：IDENTIFIER_TABLE_ALLOC_SIZE）。但如果是带参数的宏，特别是具有函数功能的部分，则要遵循函数的命名规则（如：small(a, b)）。
5. 模块中向外公开的函数，命名以模块名（大写字母）+ 下划线作为前缀（如：CRB_create_interpreter()）。
6. 模块中不对外公开的函数，如果函数的作用域跨文件时，则函数名以模块名（小写字母）+ 下划线作为前缀（如：crb_alloc_expression()）。
7. 函数外的静态变量名以st_作为前缀（如：st_string_literal_buffer）。

各模块中向外部公开的接口需要做成**公有头文件**的形式，在头文件中定义了公开函数以及调用模块所需的类型。比如 crowbar 中，想使用 crowbar 解释器就需要包含 CRB.h，而编写 crowbar 的内置函数则需要包含 CRB_dev.h。

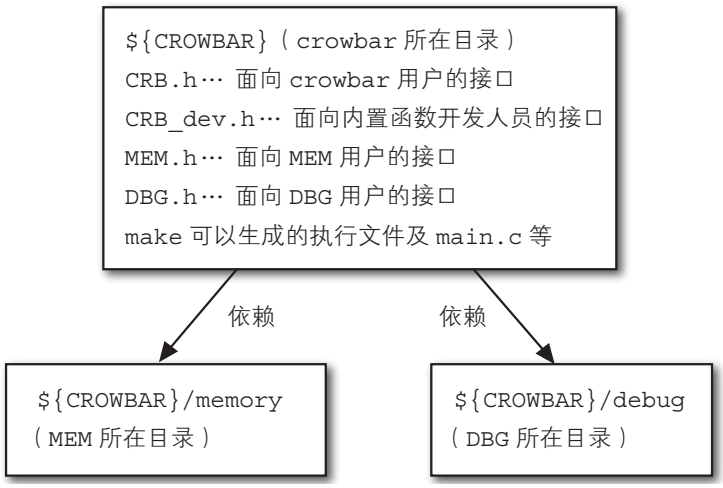
各模块内部使用的类型、宏、函数等，则可以声明为**私有头文件**。比如在 crowbar 中，crowbar.h 就是一个私有头文件，其中声明的类型名或宏无需附加 CRB_ 前缀（因为外部是接触不到的）。但是函数与全局变量，为了以防万一还是需要加上 crb_ 前缀的。

2.3.1 节后的补充知识中曾写道，所有的头文件应当尽量只用一个 #include（前提是已经加入了防止多重定义的处理）。因此大多数情况下，私有头文件内部可以用 #include 包含公有头文件，反之则不行。内部文件中使用公共信息，而外部文件中则不能含有私有信息，这应该不难理解。

经过上述的处理，各模块的内部细节都可以对其他模块实现隐藏（即面向对象中常提到的封装概念）。此外，在 C 语言中，头文件修改后包含该头文件的源代码都需要重新编译。将头文件划分为公有及私有，只要保证公有头文件不修改，那么用户利用公有头文件编写的程序也就无需重新编译了。

crowbar 的模块与目录结构如图 3-2 所示。

图 3-2
crowbar 的模块与目录结构



3.2.2 内存管理模块 MEM

经常使用 C 的程序员应该深有体会，用 C 语言编程时，难免会遇到诸如内存损坏（memory corruption）BUG、忘记释放内存引起内存泄漏、引用的内存区域被释放让 BUG 难以重现等问题，总之围绕内存经常会发生很多让人讨厌的 BUG。

而由于 crowbar 还设置有字符串型的变量，可以用 + 运算符连接字符串，因此我们必须配置某种垃圾回收机制。比如：

```
a = "a" + "b" + "c"
```

这个语句运行的时候，首先执行 "a" + "b" 语句生成字符串 "ab"，然后

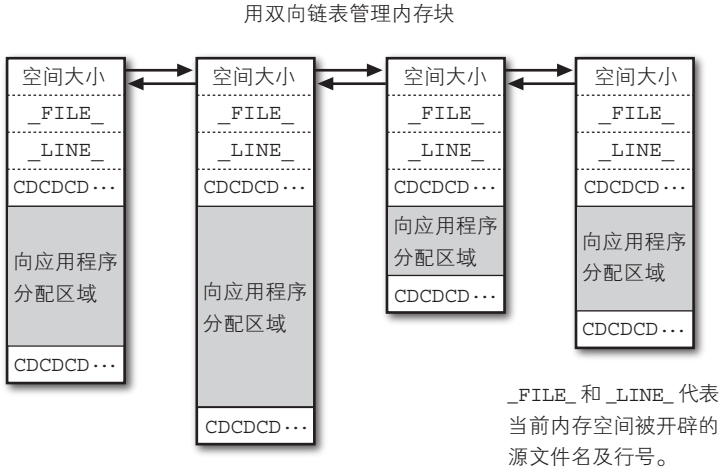
为了继续生成 "abc"，"ab" 的内存空间必须自动释放。具体的运行过程请参考 3.3.11 节。正是由于运行时需要运行很多这样繁复的处理，很容易出现 BUG，所以需要有一种方法来确认内存中到底发生了什么。

基于上述理由，我制作了一个具备下列功能的内存管理模块。模块名为 MEM，按之前的命名规范，所有的公共函数都以 MEM_ 为前缀。

1. 通过MEM_malloc()可以分配内存空间，内存空间开始处默认填充有0xCC。常规的malloc()函数开辟的内存空间值为0的情况很多，因此很容易遗漏初始化过程。而0xCC毫无疑问是个无意义的值，这样就可以确保能够检查出被遗漏的初始化过程。
2. MEM_realloc()用于扩充内存空间时，也会默认填充0xCC。
3. 开辟的内存空间用MEM_free()释放时，被填充的0xCC也会被释放。由此可以较早地发现由于引用被释放的内存空间而引起的BUG。
4. MEM模块会以链表形式保存所有开辟的内存空间，可以使用MEM_dump_block()将其转储。转储后可以将MEM_malloc调用位置的源文件名及行号显示出来。
   用malloc()开辟的内存空间，在不用的时候一定要用free()释放，这是我们在编程时一定要遵守的一个准则。那么如果在程序结束时调用MEM_dump_block()仍然看到有结果输出的话，就可以断定某处发生了内存泄漏。
5. MEM_malloc()开辟的内存空间在传递给程序使用时，空间前后会加上0xCD的记号，检查这些记号就可以知道由于数组越界等问题引起的内存损坏程度了。
   这个检查还需要配合使用MEM_check_block()、MEM_check_all_blocks()等函数。

内存管理模块 MEM 会以图 3-3 的形式管理内存。

图 3-3
通过 MEM 管理内存



很简单的模块，功能虽然简单，但对于 BUG 的检查非常有用。

对于动态开辟的内存空间，经常会先开辟若干个小型的区域，然后将这些区域一起释放。分析树的节点就是典型的例子。开辟空间会一点一点地进行，释放则是一次性的。对此，MEM 模块引入了存储器（storage），作为开辟内存的常规工具。

1. 由MEM_open_storage()生成一个新的存储器。
2. MEM_storage_malloc()可以接受存储器和空间的大小作为入口参数，并返回所请求大小的内存空间。
3. 由MEM_dispose_storage()将存储器内所有的内存空间全部释放。

MEM_storage_malloc() 会将 MEM_open_storage() 开辟的较大内存空间，从起始处按照请求的尺寸一次性全部返回。因此无法对其中的子空间单独释放，也不能通过 realloc() 扩展空间。

补充知识　valgrind

我们手动实现了模块 MEM，它可以检查由于 C 语言操作内存而引起的 BUG，其实也有很多其他工具具备同样的功能。

以前这类工具大都是需要付费的，不过在 Linux 环境下，可以使用免费软件 valgrind（许可证为 GPL）。

通常我们使用下面的方式启动程序（% 是命令行提示符）。

```
% crowbar test.crb
```

而执行如下指令的话，

```
% valgrind crowbar test.crb
```

可以帮助我们检查是否忘记释放内存（或内存泄漏），以及是否在程序开辟的内存空间外部进行了写入。

可能有读者会问，有这么方便的工具为什么还要制作 MEM 模块呢？实际上，我在写 MEM 模块时完全不知道有 valgrind 这个工具，算是重复发明轮子了。不过自己实现一个这样的工具也是有好处的吧。

valgrind 的详细内容，请参考官方主页 http://valgrind.org/。

补充知识　富翁式编程

MEM 模块中，在应用程序所使用的内存空间前后分别加上了管理专用空间。比如开发环境中 int 型或者指针一般占用 4 字节，double 型一般占用 8 字节，而其管理空间前面占用 24 字节，后面则有 8 字节（包含校验信息）。

那么如果生成很多对象时，肯定会浪费很多内存空间。

然而对于现在的电脑来说，这种程度的浪费简直是微不足道的。与其冥思苦想节省内存空间或提高处理速度的小技巧，倒不如专注于如何提高开发效率。这种编程方式就叫作**富翁式编程***。

crowbar的实现就非常之“富翁”。比如用crowbar书写的程序中会出现hoge_piyo_foo_bar这样一个变量。对于现代编程语言来说，这样长的变量名或函数名是很常见的。在程序中，hoge_piyo_foo_bar变量名可能还会出现若干次，crowbar的解释器将会预先对所有出现的变量名分配好空间，当然这都是要消耗内存的，最终只需要用strcmp()简单地对当前变量名做一致性检查就可以了。另外，在检索变量或函数时，都采用线性检索。

这样设计当然可能会出现运行速度慢的情况，即便如此，等到状况发生时再想办法优化也不迟。在前期优先考虑的应该是如何让程序更加容易编写、理解起来更加简单。

* 参考URL：http://www.pitecan.com/fugo.html

补充知识　符号表与扣留操作

刚刚提到过，无论是内存空间还是处理速度，crowbar的内部实现都是比较“富翁式”的。那么如果在此基础上想要进一步提高运行效率的话要怎样做呢?

正如上文所述，crowbar对于程序中多次出现的变量名等，会分别开辟空间将其保存。如果变量名较长时比较浪费，因此将同名变量整合为一处保存，不失为一个提高效率的方法。

具体来说，程序中会存在一个函数，为所有出现的特征符建立数据结构，新出现的特征符如果已经被记录则会返回其指针，如果尚未记录则会新录入并返回指针。这样的操作称为**扣留**(intern)。对一个标识符进行扣留操作时，无需判别该标识符是局部变量还是全局变量，或是函数名（当然进行判别也无妨）。

对程序中出现的所有的标识符一一进行扣留操作的话，在判断两个标识符是否为同一个时，只需要比较它们的指针就可以了。这比用strcmp()更快。

而crowbar对于局部变量、全局变量和函数则分别使用链表进行管理。一旦语句中出现变量名时，将从链表头部开始检索（采用线性检索）。如果要优化这个部分，可以考虑引入缓存、树或二分法查找等。刚才提到的这些数据结构及算法，都是编程语言语法处理器普遍使用的，读者可以自行查阅相关图书或网站。

一般来说，我们将编译器保存变量名、函数名的数据结构称为“符号表”。

3.2.3 调试模块DBG

DBG是调试时使用的模块，具备若干功能，在crowbar的代码中使用的话，只需要调用宏DBG_assert()及DBG_panic()即可。

```
/* 断言这里 a 的值应该为 5 */
DBG_assert(a == 5, ("a..%d", a));
```

这样书写的话，当a == 5这一条件不成立时，程序会将该处的源代码行号输出并执行abort()。第二个入口参数则是将想要输出的东西传递给printf()并格式化（因为宏无法使用可变长度的参数，因此需要从第二参数起全部用括号括起来）。

DBG的输出目标可以通过DBG_set_debug_write_fp()函数进行更改，标准输出目标是stderr。而输出目标无论如何更改，stderr仍然会保留一份同样的信息。因此如果不做任何更改的话，会看到stderr输出的是两行同样的信息。

DBG_panic()函数可以书写在一些程序不应该进入的分支处。典型的例子就是switch case的default分支，如：

```
/* 变量 operator 通过 switch case 判断分支条件
  default 分支在正常情况下不应当进入 */
default:
    DBG_panic(("bad case...%d", operator));
```

与DBG_assert()一样，用两层括号包裹，最终会通过printf()格式化输出。

DBG_assert()与DBG_panic()都是宏，只要在定义#define DBG_NO_DEBUG的状态下编译，就可以完全删除执行文件中的调试部分。

3.3 crowbar ver.0.1 的实现

预先准备已经差不多了，终于可以开始阅读crowbar book_ver.0.1的代码了。

3.3.1 crowbar 的解释器——CRB_Interpreter

一般来说，程序的数据结构要比运行流程更加重要，因此我们就从crowbar解释器所用的结构体CRB_Interpreter开始看起。

想使用crowbar，首先需要生成解释器，然后将解释器的源码传递给编译器（生成分析树），就可以运行了。

解释器的定义如下所示（位于crowbar.h）。注意CRB.h中公开的CRB_Interpreter是这个结构体的不完全定义，下面这个结构体定义本身对外是隐藏的。

```
struct CRB_Interpreter_tag {
    MEM_Storage         interpreter_storage;
    MEM_Storage         execute_storage;
    Variable            *variable;
    FunctionDefinition  *function_list;
    StatementList       *statement_list;
    int                 current_line_number;
};
```

解释器会保存以下内容：

1. 与解释器相同生命周期的MEM_Storage（interpreter_storage）

不再需要分析树时，需要将其释放，如3.2.2节所述，可以使用内存管理模块MEM提供的存储器功能来管理。

interpreter_storage存储器，在解释器生成时被生成，解释器废弃的同时被释放。通过CRB_Interpreter自己来开辟这个存储器。

该存储器在内存中的开辟，是通过位于util.c中的crb_malloc()工具函数实现的。

2. 运行时使用的MEM_Storage（execute_storage）

execute_storage是运行时使用的存储器。不过由于运行时必备的数据结构大多数都没有固定的释放顺序，因此execute_storage现阶段主要用于存放全局变量。

3. 全局变量链表（variable）

Variable结构体的定义如下所示：

```
typedef struct Variable_tag {
    char        *name;  /* 变量名 */
    CRB_Value   value;  /* 变量值 */
    struct Variable_tag *next;  /* 指向下一个变量的指针 */
} Variable;
```

首先这个结构体中有next这一成员，这是为了构建链表用的。CRB_Interpreter的成员variable保存在最开头。通过这样的链表，可以得到所有的全局变量。

crowbar中变量是在首次赋值时生成的，因此在运行时会有变量逐一进入，链表也会越来越长。

Variable结构体的name成员顾名思义会保存变量名，而value成员则会保存该变量的值，具体请参考3.3.8节对CRB_Value的说明。

4. 函数定义链表（function_list）

function_list是记录crowbar中编写函数的链表。语法解析时会创建这个function_list以及下面的statement_list。

FunctionDefinition类型的定义如下所示：

```
  typedef enum {
      CROWBAR_FUNCTION_DEFINITION = 1,  /* crowbar中定义过的函数 */
      NATIVE_FUNCTION_DEFINITION        /* 内置函数 */
  } FunctionDefinitionType;

typedef struct FunctionDefinition_tag {
    char                *name;   /* 函数名 */
    FunctionDefinitionType          type;  /* 函数的类型 */
    union {
        struct {
            ParameterList          *parameter;  /* 参数的定义 */
            Block                  *block;      /* 函数的主体 */
        } crowbar_f;
        struct {
            CRB_NativeFunctionProc     *proc; /* 后文详述 */
        } native_f;
    } u;
    struct FunctionDefinition_tag      *next;  /* 链表用 */
 } FunctionDefinition;
```

FunctionDefinition类型的type成员中，会区分crowbar定义的函数以及内置函数。crowbar定义的函数会使用下面联合体u的crowbar_f，而内置函数则会使用native_f。通过这种方法，可以让没有继承概念的C语言实现类似继承的功能（具体方法之后会慢慢提到）。

crowbar定义的函数会通过crowbar_f成员保存其函数参数及函数主体（执行语句）。ParameterList结构体如下所示，会将变量名做成链表并保存（crowbar是无类型语言，无需保存变量类型）。

```
  typedef struct ParameterList_tag {
      char        *name;               /* 变量名 */
      struct ParameterList_tag *next;  /* 链表所用指针 */
```

```
} ParameterList;
```

用block成员保存函数的执行语句。block是Block类型的结构体，Block类型的定义如下所示：

```
typedef struct {
    StatementList       *statement_list;
} Block;
```

StatementList正如其名称，是语句的链表。其结构体内容请参考第5项。

5. 语句链表（statement_list）

statement_list是语句的链表。其类型与上面的Block结构体保存时所用的StatementList类型相同。无论是函数定义{ }内的语句，还是顶层结构中的语句，从内部来讲都保存在StatementList中。

crowbar的解释器在语法解析后，顶层结构的语句，也就是statement_list最开头保存的语句会按照顺序开始执行。

6. 编译时当前的行号（current_line_number）

出现错误信息需要行号。current_line_number可以在编译时显示当前的行号。

current_line_number在编译结束后不会再使用。运行时如果发生错误当然也需要显示行号，这里的行号保存在分析树的语句节点中。

补充知识　不完全类型

CRB_Interpreter类型结构体是在crowbar的私有头文件crowbar.h中定义的。不过这是供解释器内部使用的数据结构，不应该向外部公开。

而生成解释器的函数CRB_create_interpreter()，它的原型定义则是在公有头文件CRB.h中：

```
CRB_Interpreter *CRB_create_interpreter(void);
```

CRB_create_interpreter()返回值的类型为CRB_Interpreter*，为了支持这样的原型定义必须首先定义CRB_Interpreter结构体。但是我们不能把解释器的内部定义直接拿出来放在公有头文件中。

应对这种情况可以使用**不完全类型**。公有头文件中只定义结构体的标识符，实际的定义是由私有头文件传递给公有头文件的。

比如上面的CRB_Interpreter类型，在CRB.h中可以做如下的标识符定义，并用typedef命名。

```
typedef struct CRB_Interpreter_tag CRB_Interpreter;
```

这种状态的 CRB_Interpreter 就是不完全类型。

不完全类型只能使用指针，即指向不完全类型的指针的变量无法被声明，不完全类型本身也无法声明变量，对不完全类型无法使用 sizeof。因此，我们是无法知道一个不完全类型的大小的，当然也无法引用其成员。

而 crowbar.h 是类型初始定义所在的地方，因此没有这些限制（详见 3.3.1 节）。这里的 CRB_Interpreter 类型就不是不完全类型。

上述都是 C 语言编程中必须掌握的一些技巧，但我意外地发现似乎了解的人不多，因此写了下来。

3.3.2 词法分析——crowbar.l

crowbar 的 lex 定义文件是 crowbar.l。源代码的摘录版本如代码清单 3-4 所示。

代码清单 3-4
crowbar.l

```
 1: %{
     省略 C 编码部分
19: %}
     开始条件
20: %start COMMENT STRING_LITERAL_STATE
21: %%
     保留字的定义
22: <INITIAL>"function"     return FUNCTION;
23: <INITIAL>"if"           return IF;
     省略其他的保留字定义
     符号类的定义。LP，RP 是 Left/Right Paren 的缩写
     LC，RC 是 Left/Right Curly（花括号）的缩写
35: <INITIAL>"("            return LP;
36: <INITIAL>")"            return RP;
37: <INITIAL>"{"            return LC;
38: <INITIAL>"}"            return RC;
     省略其他的符号定义
     标识符（变量名、函数名等）
55: <INITIAL>[A-Za-z_][A-Za-z_0-9]* {
56:     yylval.identifier = crb_create_identifier(yytext);
57:     return IDENTIFIER;
58: }
     数值。整数类型与实数类型分别处理，与 mycalc.y 相同
59: <INITIAL>([1-9][0-9]*)|"0" {
60:     Expression  *expression = crb_alloc_expression(INT_EXPRESSION);
61:     sscanf(yytext, "%d", &expression->u.int_value);
62:     yylval.expression = expression;
63:     return INT_LITERAL;
```

```
 64: }
 65: <INITIAL>[0-9]+\.[0-9]+ {
 66:     Expression  *expression = crb_alloc_expression(DOUBLE_EXPRESSION);
 67:     sscanf(yytext, "%lf", &expression->u.double_value);
 68:     yylval.expression = expression;
 69:     return DOUBLE_LITERAL;
 70: }
     定义字符串开始
 71: <INITIAL>\" {
 72:     crb_open_string_literal();
 73:     BEGIN STRING_LITERAL_STATE;
 74: }
 75: <INITIAL>[ \t] ;
     遇到换行符则增加行号
 76: <INITIAL>\n {increment_line_number();}
     定义注释的开始
 77: <INITIAL>#     BEGIN COMMENT;
     如果不符合上述定义，则为非法字符并报错
 78: <INITIAL>.       {
 79:     char buf[LINE_BUF_SIZE];
 80:
 81:     if (isprint(yytext[0])) {
 82:         buf[0] = yytext[0];
 83:         buf[1] = '\0';
 84:     } else {
 85:         sprintf(buf, "0x%02x", (unsigned char)yytext[0]);
 86:     }
 87:
 88:     crb_compile_error(CHARACTER_INVALID_ERR,
 89:                       STRING_MESSAGE_ARGUMENT, "bad_char", buf,
 90:                       MESSAGE_ARGUMENT_END);
 91: }
 92: <COMMENT>\n     {
 93:     increment_line_number();
 94:     BEGIN INITIAL;
 95: }
 96: <COMMENT>.      ;
 97: <STRING_LITERAL_STATE>\"        {
 98:     Expression *expression = crb_alloc_expression(STRING_EXPRESSION);
 99:     expression->u.string_value = crb_close_string_literal();
100:    yylval.expression = expression;
101:    BEGIN INITIAL;
102:    return STRING_LITERAL;
103: }
104: <STRING_LITERAL_STATE>\n        {
105:    crb_add_string_literal('\n');
106:    increment_line_number();
```

```
107: }
108: <STRING_LITERAL_STATE>\\\"     crb_add_string_literal('"');
109: <STRING_LITERAL_STATE>\\n      crb_add_string_literal('\n');
110: <STRING_LITERAL_STATE>\\t      crb_add_string_literal('\t');
111: <STRING_LITERAL_STATE>\\\\     crb_add_string_literal('\\');
112: <STRING_LITERAL_STATE>.        crb_add_string_literal(yytext[0]);
113: %%
```

crowbar.l 与之前计算器的例子 mycalc.l 相比（代码清单 2-1）要长很多，但本质上没有太大变化，只是使用了新的**开始条件**功能。与 mycalc.l 不同的是，在 crowbar.l 的大部分规则前都要书写 <INITIAL>，这就是开始条件。

在 crowbar 中，开始条件主要用于分割注释与字面常量（literal）。

crowbar 的注释由 # 开头直到行尾，可以用简单的正则表达式 #.*$ 将其分割，分割出的注释暂时保存在全局变量 yytext 中。

在以前的 lex 处理器中，给 yytext 分配了一个固定大小的 char 数组，而且数组的大小是无法扩展的。不过包含 flex 在内，最近发布的 lex 处理器中，yytext 已经更改为 char*，当一个很长的记号进入时，也可以动态扩展储存空间，注释最终也是要被丢弃的，如果还在这里特意去扩展存储空间的话，就显得有点笨了。

crowbar 的字面常量与 C 语言一样，是包含 \n 和 \t 的，还可以通过 \" 显示双引号本身，因此简单通过 \".*\" 的正则表达式规则进行匹配是不行的。

应对这种情况可以使用开始条件。在动作中书写 BEGIN COMMIT 切换 lex 的状态，其对应的规则就变成后面用 <COMMENT> 开始的部分了。lex 使用 INITIAL 定义了开始条件的初始状态。

crowbar.l 中，通过下面的处理将注释读入并丢弃。

```
77: <INITIAL>#      BEGIN COMMENT;
     中间省略
92: <COMMENT>\n     {
93:     increment_line_number();
94:     BEGIN INITIAL;
95: }
96: <COMMENT>.      ;
```

代码第 77 行，INITIAL 状态下如果有 # 进入，则转换为 COMMENT 状态。crowbar 的注释由 # 开始直至行尾，因此在 COMMENT 状态下如果遇到换行则切换回 INITIAL 状态（第 92 ～ 95 行的 increment_line_number() 会在后文详述）。所以，COMMENT 状态下会将除换行符以外的字符全部丢弃（第 96 行）。

关于字符串的字面常量处理，开始时调用crb_open_string_literal()，中间的字符通过crb_add_string_literal()追加，最后通过crb_close_string_literal()结束一个字符串的处理。

在这个过程中，字符串会被保存在string.c的st_string_literal_buffer这一static变量中。我本人对于使用static变量还是有些抵制的，但是鉴于我们只能把yacc/lex作为工具来使用，即使有什么想法也无法修改（至少老版本是不允许修改的），因此crowbar最终还是允许在编译器中使用静态变量的（请参考本节的补充知识）。

语法处理器在编译时如果发生错误，需要显示错误信息，因此错误信息中必须包含行号。

在crowbar.l中的对应处理是，每换行一次，行号都会进行计数。具体来说有以下三处：

```
 76: <INITIAL>\n {increment_line_number();}
   ⋮
 92: <COMMENT>\n     {
 93:     increment_line_number();
 94:     BEGIN INITIAL;
 95: }
   ⋮
104: <STRING_LITERAL_STATE>\n        {
105:     crb_add_string_literal('\n');
106:     increment_line_number();
107: }
```

这里进行计数的行号保存在CRB_Interpreter的current_line_number中。

补充知识　静态变量的许可范围

如上文所写，在词法分析中，正在读入的字符串会保存在string.c的st_string_literal_buffer中。在编译时，当前的编译器会保存在util.c的st_current_interpreter中。而在yacc/lex中，还会使用yytext等全局变量。

使用如此多的静态变量，首当其冲会遇到的就是多线程问题。

当下多线程的程序已经很普及了，静态变量可以在多线程之间共享，因此多个线程如果同时进行编译的话可能会引发问题。

不过一般来说，编译过程都是一下子就能结束的，因此在这样短的时间内通过加一个全局锁的方式就可以解决问题了。正因为有这样既简单又实用的方法，crowbar才放心地允许在编译过程中使用静态变量。而静态变量仅在编译过程中被使用，一旦程序开

始运行后就无法使用了。具体来说，由于 MEM 模块会静态地保存内存块链表，这原本就是以调试为目的创建的链表，可以随时删除，而由于 MEM 模块位于系统最底层，因此可以对 MEM_malloc() 的运行进行加锁处理。

使用静态变量还会造成另一个问题，就是编译器无法递归运行。比如 crowbar 中的库函数都书写在独立文件中，在 C 语言中理论上只需要用 #include 就能很简单地把功能包含进来。但是实际应用中就会发现，使用 #include 包括进来的独立文件，在编译过程中如果再开始一个解析器，之前的静态变量就会被覆盖。

标准版的 yacc/lex，由于使用了 yytext 等全局变量，因此也无法支持多线程或使用递归。不过 bison 有单独的扩展可以让其支持多线程。*

* 在后面写到的 Diksam 中，具备与 C 语言的 #include 相对应的功能 require，同样由于解析器不能进行递归，会在解析完一个文件后，才开始解析被 require 的文件。

3.3.3 分析树的构建——crowbar.y 与 create.c

crowbar 中 yacc 的定义文件为 crowbar.y。从构成来说，与计算器版的 mycalc.y 没有什么变化。

但是在计算器中，归约是在实际进行计算时才进行的，而 crowbar.y 则是在构建分析树时进行的。

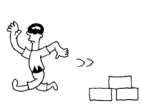
比如一个加法算式（如 10 + a）会按照以下的规则构建：

```
additive_expression
    （中间省略）
    | additive_expression ADD multiplicative_expression
    {
        $$ = crb_create_binary_expression(ADD_EXPRESSION, $1, $3);
    }
```

这里的 additive_expression 对应 mycalc.y 中的 expression，multiplicative_expression 对应 term。

在动作中，crb_create_binary_expression() 被调用，其实际运行代码在 create.c 中。这个函数负责常量折叠（参考 3.3.4 节），所以稍微有些复杂，将这部分以外的核心代码精简一下，可以看到这个函数的主要逻辑如下所示：

```
Expression *
crb_create_binary_expression(ExpressionType operator,
                             Expression *left, Expression *right)
{
    Expression *exp;
    exp = crb_alloc_expression(operator);
    exp->u.binary_expression.left = left;
    exp->u.binary_expression.right = right;
```



```
    return exp;
}
```

crb_alloc_expression() 将开辟一个存放 Expression 类型结构体的内存空间，并将其返回。Expression 类型在 crowbar.h 中的定义如下所示：

```
struct Expression_tag {
    ExpressionType type;    ←表示表达式的类别
    int line_number;
    union {                 ←用联合体保存不同种类对应的值
        CRB_Boolean             boolean_value;
        int                     int_value;
        double                  double_value;
        char                    *string_value;
        char                    *identifier;
        AssignExpression        assign_expression;
        BinaryExpression        binary_expression;
        Expression              *minus_expression;
        FunctionCallExpression  function_call_expression;
    } u;
};
```

这个结构体在分析树中用来表示“表达式”的类型。与 CRB_Value 一样，用枚举类型的 ExpressionType 表示表达式的类型，用联合体保存各种类型对应的值。

ExpressionType 具体定义如下：

```
typedef enum {
    BOOLEAN_EXPRESSION = 1,     /* 布尔型常量 */
    INT_EXPRESSION,             /* 整数型常量 */
    DOUBLE_EXPRESSION,          /* 实数型常量 */
    STRING_EXPRESSION,          /* 字符串型常量 */
    IDENTIFIER_EXPRESSION,      /* 变量 */
    ASSIGN_EXPRESSION,          /* 赋值表达式 */
    ADD_EXPRESSION,             /* 加法表达式 */
    SUB_EXPRESSION,             /* 减法表达式 */
    MUL_EXPRESSION,             /* 乘法表达式 */
    DIV_EXPRESSION,             /* 除法表达式 */
    MOD_EXPRESSION,             /* 求余表达式 */
    EQ_EXPRESSION,              /* == */
    NE_EXPRESSION,              /* != */
    GT_EXPRESSION,              /* > */
    GE_EXPRESSION,              /* >= */
    LT_EXPRESSION,              /* <  */
    LE_EXPRESSION,              /* <= */
    LOGICAL_AND_EXPRESSION,     /* && */
```

```
    LOGICAL_OR_EXPRESSION,      /* || */
    MINUS_EXPRESSION,           /* 单目取负 */
    FUNCTION_CALL_EXPRESSION,   /* 函数调用表达式 */
    NULL_EXPRESSION,            /* null 表达式 */
    EXPRESSION_TYPE_COUNT_PLUS_1
} ExpressionType;
```

这其中从ADD_EXPRESSION到LOGICAL_OR_EXPRESSION，都在使用联合体的binary_expression成员。

binary_expression的类型是BinaryExpression，其定义如下所示：

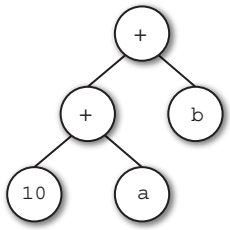
```
typedef struct {
    Expression  *left;
    Expression  *right;
} BinaryExpression;
```

crb_create_binary_expression()最终返回这样构建出的Expression的指针，并在crowbar.y中将其赋值给$$。

```
$$ = crb_create_binary_expression(ADD_EXPRESSION, $1, $3);
```

比如10 + a + b这样的语句，按上面的处理就会构建出如图3-4的分析树。

图3-4
10+a+b的分析树



那么，接下来是crb_alloc_expression()函数，这个函数只是简单地用crb_malloc()开辟Expression的空间，并且将type连同刚被设置的行号一起返回。

```
crb_alloc_expression(ExpressionType type)
{
    Expression  *exp;

    exp = crb_malloc(sizeof(Expression));
    exp->type = type;
    exp->line_number = crb_get_current_interpreter()->current_line_number;

    return exp;
}
```

在分析树中，不单是表达式，语句（statement）也很重要。构建语句的思路与表达式基本是一样的，关联的结构体定义从 crowbar.h 中摘录如下：

```
struct Statement_tag {
    StatementType       type;
    int                 line_number;
    union {
        Expression      *expression_s;  /* 表达式语句 */
        GlobalStatement global_s;       /* global 语句 */
        IfStatement     if_s;           /* if 语句 */
        WhileStatement  while_s;        /* while 语句 */
        ForStatement    for_s;          /* for 语句 */
        ReturnStatement return_s;       /* return 语句 */
    } u;
};
```

随便举一个联合体的例子，比如 WhileStatement 的定义如下所示：

```
typedef struct {
    Expression  *condition;  /* 条件表达式 */
    Block       *block;      /* 可执行块 */
} WhileStatement;
```

Block 在 3.3.1 节中已经出现过一次，即一段用花括号包裹的代码段类型。

StatementType 的一览表如下所示：

```
typedef enum {
    EXPRESSION_STATEMENT = 1,
    GLOBAL_STATEMENT,
    IF_STATEMENT,
    WHILE_STATEMENT,
    FOR_STATEMENT,
    RETURN_STATEMENT,
    BREAK_STATEMENT,
    CONTINUE_STATEMENT,
    STATEMENT_TYPE_COUNT_PLUS_1
} StatementType;
```

BREAK_STATEMENT 和 CONTINUE_STATEMENT 现阶段都没有信息需要保存（如果想像 Java 那样支持标签或 break 语句的话就需要保存了），因此没有对应的联合体成员。

3.3.4 常量折叠

比如

```
a = a + 10 * 2;
```

这样一个表达式，其实与

```
a = a + 20;
```

是同值的。诸如这种纯常量构成的表达式或部分表达式，在编译时提前被计算出来的处理方式叫作**常量折叠**(constant folding)。由于编译时这部分计算就已经完成了，所以能有效地提高运行速度。

crowbar 中也部分引入了常量折叠的处理。具体来说，在进行整数型与实数型相关的四则运算或者单目取负时会进行常量折叠 *。这部分代码本书不会进行详细说明，请参考 create.c 的 `crb_create_binary_expression()` 和 `crb_create_minus_expression()`。

* 字符串的加法不会做折叠处理。确实代码中有时候需要有长文本信息，但是按当前版本的处理方式，连接前的字符串是无法被释放的，也就是说其实这里是我偷懒了。

crowbar 中的常量折叠其实可以归入所谓程序最优化（ optimization ）的范畴，在语言的制作阶段考虑最优化本来为时尚早，但是由于 crowbar 的目标是类似 C 的语言，这种优化在部分场景中是必须的。比如 C 语言在一些特定地方只能写常量表达式（ `static` 变量的初始化或数组的大小指定等 * ），这些地方需要支持常量折叠的处理。

* ISO-C99 规范中，auto 的数组也可以不是常量。

程序优化是指通过不断调整程序以提高代码的运行速度，但是调整的结果到底是不是“最优”其实不好评判，因此有些人提出“最优化”这个用词是不恰当的。

3.3.5 错误信息

错误信息考虑到需要支持多语言 *，所以一般尽可能避免硬编码出现在代码中，最好以提供外部文件的方式实现。但是在开始设计 crowbar 这样的脚本语言时，我非常希望 crowbar 只通过一个可执行文件就能运行。比如我想去别的地方做一些文字处理工作，那么只需要把 crowbar 的唯一一个可执行文件拷入 U 盘就可以拿去用了。*

* 可能很多读者会认为显示错误信息只有英语就足够了，不过对于初级用户来说，英文的错误信息可能有点难以理解吧。更重要的理由是：我觉得自己的英语水平还不够。

* 不过一般公司对于数据的管理都是比较严格的，可能没有办法用 U 盘携带数据吧。

因此 crowbar 还是选择将错误信息硬编码在 error_message.c 的源文件中，参看代码清单 3-5。

代码清单 3-5
error_message.c

```
MessageFormat crb_compile_error_message_format[] = {
    { "dummy"},
    { "($(token) 附近有语法错误 )"},
    { " 错误的字符 ($(bad_char))"},
    { " 函数名重复 ($(name))"},
    { "dummy"},
};

MessageFormat crb_runtime_error_message_format[] = {
    { "dummy"},
    { " 找不到变量 ($(name))。"},
    { " 找不到函数 ($(name))。"},
    { " 传递的参数多于函数所要求的参数。"},
    { " 传递的参数少于函数所要求的参数。"},
    { " 条件语句类型必须为 boolean 型 "},
    ⋮
    { "dummy"},
};
```

通过这种方式，将来如果要支持多语言的话，可以简单地通过修改外部文件来实现。而设置 crb_compile_error_message 与 crb_runtime_error_message 这两个数组，是为了将编译时的错误信息与运行时的错误信息区分开来。这些数组的索引，与 crowbar.h 中对应的枚举型的值是一致的。

代码清单 3-6
crowbar.h 的错误信息枚举型定义

```
typedef enum {
    PARSE_ERR = 1,
    CHARACTER_INVALID_ERR,
    FUNCTION_MULTIPLE_DEFINE_ERR,
    COMPILE_ERROR_COUNT_PLUS_1
} CompileError;

typedef enum {
    VARIABLE_NOT_FOUND_ERR = 1,
    FUNCTION_NOT_FOUND_ERR,
    ARGUMENT_TOO_MANY_ERR,
    ⋮
    RUNTIME_ERROR_COUNT_PLUS_1
} RuntimeError;
```

代码清单 3-6 中的错误信息，包含了一个 $(token) 这样的字符串，这是错误信息中的可变部分。比如找不到一个名为 hoge 的变量时，如果能报出“找不

到变量 (hoge)”，要比只报“找不到变量”对用户更加友好。

为了实现错误信息中可以包含变量，显示错误信息时会调用下面的函数 crb_runtime_error()*。

* 只适用于运行出错。如果是编译出错，则调用 crb_compile_error()。

```
crb_runtime_error(expr->line_number,  /* 行号 */
                  VARIABLE_NOT_FOUND_ERR,  /* 枚举错误信息类别 */
                  STRING_MESSAGE_ARGUMENT, /* 可变部分的类型 */
                  "name", /* 可变部分的标识符 */
                  expr->u.identifier, /* 所要显示的值 */
                  MESSAGE_ARGUMENT_END);
```

第一个参数是行号（在 create.c 中 Expression 结构体中设定），下一个参数是错误类别（crowbar.h 中的 RuntimeError 枚举型），之后的三个参数为一组，对应错误信息的可变部分。STRING_MESSAGE_ARGUMENT 代表信息类型（相当于 printf() 中使用的 %s），name 即错误信息中的 $(name)，expr->u.identifier 则表示可变部分要显示的字符串。由于可变部分可以有多个，因此 crb_runtime_error 为一个变长参数，所以可以用 MESSAGE_ARGUMENT_END 表示参数输入结束。

当前版本无论是编译错误还是运行错误，显示错误信息后都会立即调用 exit() 终止程序。这样的处理其实还远远不够，如果用于扩展应用程序的话这样做更是致命的，因此应当参考 9.2.1 节加入异常处理机制。

补充知识　关于 crowbar 中使用的枚举型定义

比如在代码清单 3-6 中的 CompileError 类型，我特意将第一个元素 PARSE_ERR 设置为 1，而最后一个元素引入了名为 COMPILE_ERROR_COUNT_PLUS_1 的可变元素。

```
typedef enum {
    PARSE_ERR = 1,                  ←特意设置为1
    CHARACTER_INVALID_ERR,
    FUNCTION_MULTIPLE_DEFINE_ERR,
    COMPILE_ERROR_COUNT_PLUS_1      ←可变元素
} CompileError;
```

类似这样的处理方式不只有 CompileError 类型。比如用于显示 Expression 类别的 ExpressionType 类型也采用同样的构造。

特意这样设置的理由有下面几个。

1. 假如忘记进行初始化时，变量中被置入 0 的概率是非常高的，那么枚举类型如果从 1 开始的话，可以更早地发现异常状态。

2. 有了 COMPILE_ERROR_COUNT_PLUS_1 这个可变元素，就可以借助其遍历所有枚举元素，并在后续程序中利用这一特性进行更丰富的处理。

当然实际使用时，我发现这两处设置似乎也不是特别有效。

另外在错误信息中，错误信息数组的第一个和最后一个元素都是 dummy，这是为了防止在以后修改时只改了 crowbar.h 中的枚举类型，而忘记修改 error_message.c 中的对应错误信息，这样设置的话能在一定程度上自动检测这种遗漏（请参考 error.c 中的 self_check() 函数）。

```
MessageFormat crb_compile_error_message_format[] = {
    {"dummy"},
    ⋮
    {"dummy"},
};
```

3.3.6 运行——execute.c

crowbar 程序的运行是从 CRB_interpret() 开始的，其函数实现如下：

```
void
CRB_interpret(CRB_Interpreter *interpreter)
{
    interpreter->execute_storage = MEM_open_storage(0);
    crb_add_std_fp(interpreter);
    crb_execute_statement_list(interpreter, NULL, interpreter->statement_list);
}
```

第一行准备了运行时要用的 MEM_Storage，第二行的函数 crb_add_std_fp() 注册了三个内部全局变量 STDIN、STDOUT 和 STDERR，然后通过 crb_execute_statement_list() 将解释器中保存的语句链表按顺序执行。

那么我们就来看一看 crb_execute_statement_list() 函数（代码清单 3-7）。

代码清单 3-7
crb_execute_statement_list()

```
StatementResult
crb_execute_statement_list(CRB_Interpreter *inter, LocalEnvironment
    *env, StatementList *list)
{
    StatementList *pos;
    StatementResult result;

    result.type = NORMAL_STATEMENT_RESULT;
    for (pos = list; pos; pos = pos->next) {
```

```
            result = execute_statement(inter, env, pos->statement);
            if (result.type != NORMAL_STATEMENT_RESULT)
                goto FUNC_END;
        }

    FUNC_END:
        return result;
    }
```

即按照链表的顺序，调用execute_statement()。

execute_statement()则会根据不同的Statement类型执行不同的处理（代码清单3-8）。

代码清单3-8
execute_statement()

```
static StatementResult
execute_statement(CRB_Interpreter *inter, LocalEnvironment *env,
                  Statement *statement)
{
    StatementResult result;

    result.type = NORMAL_STATEMENT_RESULT;

    switch (statement->type) {
    case EXPRESSION_STATEMENT:
        crb_eval_expression(inter, env, statement->u.expression_s);
        break;
    case GLOBAL_STATEMENT:
        result = execute_global_statement(inter, env, statement);
        break;
    case IF_STATEMENT:
        result = execute_if_statement(inter, env, statement);
        break;
    case WHILE_STATEMENT:
        result = execute_while_statement(inter, env, statement);
        break;
    ⋮
    case STATEMENT_TYPE_COUNT_PLUS_1:   /* FALLTHRU */
    default:
        DBG_panic(("bad case...%d", statement->type));
    }

    return result;
}
```

execute_statement()的第二个参数需要传递LocalEnvironment类型的结构体（指向其的指针）。这个结构体保存了当前运行中的函数的局部变量，

如果函数还没有运行，则传递NULL。

execute_statement()内部采用了switch case来区分条件处理。如果将其调用的函数全部进行分析的话有点浪费篇幅了，这里以while语句调用的execute_while_statement()为代表进行说明（代码清单3-9，移除了错误检查等与核心功能无关的代码）：

代码清单3-9
execute_while_statement()

```
static StatementResult
execute_while_statement(CRB_Interpreter *inter, LocalEnvironment
    *env, Statement *statement)
{
    StatementResult result;
    CRB_Value   cond;

    result.type = NORMAL_STATEMENT_RESULT;
    for (;;) {  /* 首先是一个无限循环 */
        /* 通过条件语句判别 */
        cond = crb_eval_expression(inter, env, statement->u.while_
            s.condition);
        /* 条件为真则结束循环 */
        if (!cond.u.boolean_value)
            break;

        /* 条件不为真则执行内部语句 */
        result = crb_execute_statement_list(inter, env,
                                            statement->u.while_s.block
                                            ->statement_list);

        /* break, continue, return的处理 */
        if (result.type == RETURN_STATEMENT_RESULT) {
            break;
        } else if (result.type == BREAK_STATEMENT_RESULT) {
            result.type = NORMAL_STATEMENT_RESULT;
            break;
        }
    }

    return result;
}
```

if语句或while语句都会包含一些内部的语句，这些内部语句被称为嵌套（nest）。在crowbar中，如果存在嵌套，则不会进行递归，而是转向运行嵌套内的语句。

事实上，实现上面的机制主要用到的是break、continue、return等，从

某种程度上来说用 goto 实现结构控制反而非常麻烦。

break、continue、return 等出现时，必须从递归的最深处强制返回上层。

break 或 continue 的作用是从最内层的循环中跳出，当然 break 或 continue 很可能会出现在很多层嵌套的 if 语句中，而无论其出现在哪里，正在进行的递归都必须从最底层一次性返回，这是不会改变的。

为了实现这一点，我们有一个“必杀技”可以使用——setjmp()/longjmp()*。这个必杀技还被应用到异常处理机制中，因此在 crowbar 中，返回值与结束状态会逐步返回。

* Ruby 的设计中也使用了这种必杀技。

execute_statement() 以及内部被调用的函数群 execute_XXX_statement()，返回值均为 StatementResult 的结构体。StatementResult 的定义如下：

```
typedef enum {
    NORMAL_STATEMENT_RESULT = 1,
    RETURN_STATEMENT_RESULT,
    BREAK_STATEMENT_RESULT,
    CONTINUE_STATEMENT_RESULT,
    STATEMENT_RESULT_TYPE_COUNT_PLUS_1
} StatementResultType;

typedef struct {
    StatementResultType type;
    union {
        CRB_Value       return_value;
    } u;
} StatementResult;
```

通常，type 会返回一个装入 NORMAL_STATEMENT_RESULT 的 StatementResult，而当执行 return、break、continue 时，则分别在 RETURN_STATEMENT_RESULT、BREAK_STATEMENT_RESULT、CONTINUE_STATEMENT_RESULT 装入对应的 StatementResult 并返回。此外，在代码清单 3-9 的 execute_while_statement() 等中会根据执行语句的返回值不同而所有不同，如果是 BREAK_STATEMENT_RESULT 或 RETURN_STATEMENT_RESULT 则会中断循环，而如果是 continue 的话，只会中断 crb_execute_statement_list() 中的运行。

如果是 return，那么不只要中断函数的运行，还要携带其返回值返回，此时会将返回值放入 StatementResult 的 return_value 中。

编程语言实用化指南——写在最后

在本书中，我们一起制作了 crowbar 和 Diksam 两种编程语言。让我感到欣慰的是，书中的示例程序不只是停留在入门级别，而是达到了实用语言的水准。

亲爱的读者朋友们，希望你们也能尝试制作属于自己的编程语言。不过说出来你们可能会大跌眼镜，**编程语言的魅力基本上是由它的程序库来决定的**，而这是不容争辩的事实。

例如，Perl 由于正则表达式等强有力的字符串处理功能得到了广泛的应用。在 Perl4 的时候，作为编程语言，Perl 既没有引用，也不能创建数据结构，可以说很难用。但是，因为它处理文本文件十分方便，所以得到了广泛使用。同样，PHP 也因为提供了很多面向网页应用的功能而得到了普及。语言是否实用，是否能够普及，实际上和语言的设计本身没有太大关系。

因此，我在发明了 crowbar 和 Diksam 两种语言后，为它们加载了各自的程序库。

首先，我为 crowbar 加载了鬼车，使它具有用正则表达式处理文本的能力。

在 Diksam 中我用 crowbar 来处理文本。在文本处理这个领域里已经有了 Perl、Ruby 等语言，因此就算是为此特地制作一门语言也不会得到广泛普及。用来开发 Web 应用的编程语言更是琳琅满目，比如 PHP、Perl、Ruby、Java、ASP、ASP.NET 等，在这个领域中还充斥着各种框架，可以说是一个大杂烩。租赁服务器更是让人头疼，好不容易做的网页应用，有可能会因为服务器不能支持而不能使用。就这点来说，已经很让人沮丧了。

因此，我考虑让 Diksam 定位为“让初学者可以制作简单游戏的语言”。

在很早之前，我自己就是这样走上了编程的道路。

那个时候（1980 年左右）的个人电脑，大多将 BASIC 作为标准配置。那个时候的编程语言没有 `IF` 语句中 `begin~end` 或者 {} 之类的程序块的概念，选择分支的时候必须使用 `GOTO` 行号的方式进行跳转。也没有循环结构的 `FOR` 语句。要在循环外面记录循环计数器后，再使用 `GOTO` 进行跳转。虽然可以使用 `GOSUB` 制作子程序，但是不能定义局部变量，所有变量都要当做全局变量来处理。此外，变量名字不看到第 2 个字符，是区分不出来的 *。当然，这是时代的选择，不过，当时的 BASIC 作为编程语言来说还真是不怎么样。

* Visual Basic 的名字虽然继承了 Basic，但其实是完全不同的另一种语言。

即使如此，我当时只用了几十行代码就可以写一款射击游戏（用字符“⊥”

当做炮台来击落用字符“-o-”做成的飞碟）。我觉得这个过程十分有趣，这也是我踏上编程学习道路的第一步。

但是对于现在的年轻人来说，却不知道怎么去实现一款简单的游戏。

现在这些 PC 的性能与当时相比可以说有了飞跃性的提高，也出现了各种各样的编程语言和免费的处理器。但是，比如在 C 语言中，使用不依赖处理器的标准 C，就连窗体都打不开。即便只是在 Windows 中能够运行起来的程序，C 语言也要通过 Windows 的 API 来创建窗口，如此复杂的程序初学者根本应付不来。

我觉得在现在 C 语言的入门书中，多半从“`hello,world.`”开始介绍许多命令行程序。但是，我在中学时代，从最开始就能写出和“`hello,world.`”差不多的程序，接着就编了猜数字游戏，然后就想着要做一款打飞碟的游戏了。当今，计算机已经十分先进，但人们在这个方面反而退化了。

当然，现在不仅可以使用 C 语言，也考虑使用 Java。Java 中的 GUI 可以不依赖于处理器，因此也可以把游戏做成 Applet 发布在网站上，在朋友面前炫耀一把。但这样一来，在创建类的时候就需要继承一种叫作 `java.applet.Applet` 的类，并重写它的 `init()` 和 `paint()` 之类的方法。这里突然出现了很多面向对象的知识，初学者一时之间很难接受。也许有人会觉得我又在这里老调重弹了，但是仔细想想，为了从 GUI 接收输入，就必须要创建事件监听器、实现特定的接口，除此之外还需要使用内部类和匿名类。这些还没完，因为制作的是实时游戏，为了实现动画效果还需要使用多线程进行异步处理。这些对于一个新手来说简直是个噩梦。

再者，制作“打飞碟”这样一款游戏对于 JavaScript 来说也不是很容易。可以制作 FLASH 的语言 ActionScript，它的标准处理器又不是免费的。

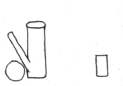
HSP（Hot Soup Processor）语言是我在中学时代玩过的类似于 BASIC 的语言。不过，很对不起这门语言的 fans，这门语言本身和与它同时代的 BASIC 如出一辙，因此对于刚开始学习编程的人来说非常不推荐。它甚至没有 GOSUB*。

因此，我在 Diksam 中加载了可以让“打飞碟”游戏实现起来更为简单的程序库 *。

因为要制作的游戏非常简单，所以无须特意想着面向对象和事件驱动的概念。例如“打飞碟”游戏可以写成下面这样。

* 从 HSP3 开始可以通过函数实现。

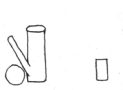
* 这种情况仅限于 Windows 平台。



代码清单 1-4
“打飞碟”游戏的程序
(ufo.dkm)

```
 1: require diksam.window;
 2:
 3: // 创建设置窗体属性的 WindowAttribute 对象。
 4: WindowAttribute attr = create_window_attribute();
 5: // 设置背景色为黑色。
 6: attr.background = create_solid_brush(0, 0, 0);
 7:
 8: // 创建窗体。如果使用默认设置就可以，
 9: // 那么不创建 attr 传入 null 即可。
10: Window w = create_window( “UFO 游戏”, 800, 600, attr);
11:
12: // 使用 x 键终止程序。
13: w.set_destroy_proc(window_destroy_and_exit);
14: // 显示窗体。
15: w.show();
16:
17: // 为了描绘窗体取得 Graphics 接口。
18: Graphic g = w.get_graphics();
19: // 将字符的背景色设置为黑色。
20: g.set_background_color(new Color(0, 0, 0));
21:
22: // 战车、激光、UFO 的颜色设置。
23: Color tank_color = new Color(60, 255, 100);
24: Color ufo_color = new Color(60, 255, 255);
25: Color beam_color = new Color(255, 255, 100);
26: // 生成字体。详细的设置（FontAttribute）
27: // 与 WindowAttribute 相同，当前默认为 null。
28: Font font = create_font(25, null);
29:
30: // 随机数的初始化
31: randomize();
32:
33: // 游戏结束后再开始使用的循环
34: for(;;){
35:     // 设定炮台（tank）、ufo 的坐标。将 tank_x，ufo_x，ufo_y 的
36:     // 当前坐标赋值给 prev，作为前一次绘制的坐标（消除时使用）。
37:     // ufo_next_x,y 作为 ufo 的移动目标的坐标。
38:     // ufo 将在 ufo_next_x,y 的附近移动，
39:     // 但如果两次坐标基本相同，则重新设定 ufo_next_x,y。
40:     int tank_x = 0;
41:     int ufo_x = 0;
42:     int ufo_y = 0;
43:     int ufo_prev_x = ufo_x;
44:     int ufo_prev_y = ufo_y;
45:     int ufo_next_x = random(700);
46:     int ufo_next_y = random(450);
47:     // 是否存在炮台发射的激光的标识和激光坐标
```

```
48:     boolean beam_flag = false;
49:     int beam_prev_y;
50:     int beam_x;
51:     int beam_y;
52:
53:     // 游戏的主循环
54:     for(;;){
55:         // 消除前一次画出来的 UFO。
56:         g.draw_string(font, ufo_color, ufo_prev_x, ufp_prev_y,"    ");
57:         // 绘制 UFO。
58:         g.draw_string(font, ufo_color, ufo_x, ufp_y," ├O┤ ");
59:         // 为了再次消除，保存本次绘制的坐标。
60:         ufo_prev_x = ufo_x;
61:         ufo_prev_y = ufo_y;
62:         // 绘制炮台。
63:         g.draw_string(font, ufo_color, tank_x, 540," /┴\ ");
64:         // 发射了激光的话……
65:         if(beam_flag){
66:             // 消除前面的激光，重画新的激光。
67:             g.draw_string(font, ufo_color, ufo_prev_x, ufp_prev_y," ");
68:             g.draw_string(font, ufo_color, ufo_x, ufp_y,"|");
69:             beam_prev_y = beam_y;
70:
71:             // 碰撞判断。可能很幼稚。
72:             if(beam_x >= ufo_x && beam_x < ufo_x + 80
73:                 && beam_y >= ufo_y && beam_y < ufo_y +60){
74:                 // 被激光打中后跳出循环。
75:                 break;
76:             }
77:         }
78:
79:         // 通过判断键盘输入移动炮台。
80:         if(is_key_pressed(KeyCode.LEFT) && tank_x > 0){
81:             tank_x -= 10;
82:         } elseif(is_key_pressed(KeyCode.RIGHT) && tank_x < 700){
83:             tank_x += 10;
84:         }
85:         if(is_key_pressed(KeyCode.SPACE) && !beam_flag){
86:             beam_flag = true;
87:             beam_x = tank_x + 40;
88:             beam_y = beam_prev_y = 480;
89:         }
90:         // UFO 的移动。在 ufo_next_x,y 的附近移动。
91:         if(ufo_x < ufo_next_x -10){
92:             ufo_x += 10;
93:         } elseif(ufo_x > ufo_next_x + 10){
94:             ufo_x -= 10;
```



```
        } else {
            // 如果两次坐标基本相同，则重新设定目标坐标。
            ufo_next_x = random(700);
        }
        if(ufo_y < ufo_next_y - 10) {
            ufo_y += 10;
        } elseif (ufo_y > ufo_next_y + 10) {
            ufo_y -= 10;
        } else {
            ufo_next_y = random(450);
        }
        // 激光的移动
        if(beam_flag) {
            if(beam_y < -20){
                beam_flag = false;
            }
            beam_y -= 20;
        }
        // 定时消息循环。无论有没有
        // 鼠标或者键盘事件，都等待 20 毫秒。
        timed_message_loop(w, 20);
    }

    // 被激光打中后跳出循环，执行这里。
    for(;;){
        // 显示爆炸效果
        Color explosion_color = new Color(255, 0, 0);
        g.draw_string(font, explosion_color, ufo_x, ufo_y, "***");
        timed_message_loop(w, 100);
        g.draw_string(font, explosion_color, ufo_x, ufo_y, "###");
        timed_message_loop(w, 100);
        // 按 N 重启游戏，按 Q 退出。
        if(is_key_pressed(KeyCode.N)){
            Brush b = create_solic_brush(0, 0, 0);
            g.fill_rectangle(b, 0, 0, 800, 600);
            b.dispose();
            break;
        } elsif (is_key_pressed(KeyCode.Q)) {
            exit(0);
        }
    }
}
```

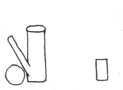
游戏的截屏如下所示。

这是个跟我同时代的人都会怀念的画面吧 *。

Diksam 在这个领域的进化旅程才刚开始，谁也不知道它在未来会变成什么样子。但是，在代码清单“ufo.dkm”的程序中，只能有一架 UFO，在同一时间炮台只能发射一发激光（因为表示 UFO 和激光位置的变量只有一组）。如果觉得这样没意思的话，就必须要使用数组了。x 坐标和 y 坐标要是都使用数组来管理的话，肯定会很不方便，如果有类的话感觉就会方便很多。同样，即使不同时出现多个飞碟，如果想要各种各样的敌人轮番登场，就要使用继承和多态了……一门语言因为追寻着这样的思路而具有了面向对象的概念，真让人兴奋。

各位读者朋友，你们想让自己的编程语言向哪个方向发展呢？希望本书能给各位带来一些启发。

* 当然，在 Diksam 中也可以很简单地使用图标来表示 UFO。



参考文献

■ 书中引用到的文献

[1] 情報処理シリーズ 7　コンパイラ
英文版：*Principles of Compiler Design*[①]

[2] プログラミング言語 C
英文版：*The C Programming Language*
中文版：《C 程序设计语言》（机械工业出版社）

[3] The Problem with Integer Division
http://python-history.blogspot.com/2009/03/problem-with-integer-division.html

[4] 文字符号の歴史―欧米と日本編
中文译名：《文字符号的历史——欧美与日本篇》
作者：安冈孝一 安冈素子
出版社：共立出版社，2006 年

[5] プログラミング作法
英文版：*The Practice of Programming*
中文版：《程序设计实践》（机械工业出版社）

[6] YARV アーキテクチャ
英文版：*YARV Architecture*
http://www.atdot.net/yarv/yarvarch.en.html

[7] The Java® Language Specification
http://docs.oracle.com/javase/specs/jls/se7/html/index.html

① *Principles of Compiler Design* 的封面是一名骑士和一只恐龙，因此被人称为“龙书”，但因为那条龙是绿色的，所以称为“绿龙书”。9 年后（1986 年），原来的两位作者加上 Ravi Sethi, 升级了这本书，书名改为 *Compilers: Principles, Techniques and Tools*，封面依然沿用骑士和恐龙，那头龙是红色的，因此被叫作“龙书二”或者是“红龙书”。——译者注

[8] オブジェクト指向入門 第 2 版 原則・コンセプト
英文版：*Object-Oriented Software Construction,Second Edition*

[9] Effective Java プログラミング言語ガイド
英文版：*Effective Java*
中文版：《Effective Java 中文版（第 2 版）》（机械工业出版社）

[10] オブジェクト指向における再利用のためのデザインパターン
英文版：*Design Patterns: Elements of Reusable Object-Oriented Software*
中文版：《设计模式：可复用面向对象软件的基础》（机械工业出版社）

[11] The Trouble with Checked Exceptions
http://www.artima.com/intv/handcuffs.html

[12] Java の理論と実践：例外をめぐる議論 チェックすべきか、チャックせずにおくべきか
中文译名：Java 理论与实践：关于异常的争论 要检查，还是不要检查?
http://www.ibm.com/developerworks/cn/java/j-jtp05254/

[13] Exceptions
http://joelonsoftware.com/items/2003/10/13.html

[14] Cleaner, more elegant, and harder to recognize
http://blogs.msdn.com/b/oldnewthing/archive/2005/01/14/352949.aspx

[15] *More Joel On Software*
中文版：《软件随想录：程序员部落酋长 Joel 谈软件》（人民邮电出版社）

[16] Python リファレンスマニュアル 2.4.1. 文字列リテラル
英文版：*The Python Language Reference 2.4.1. String literals*
http://docs.python.org/2/reference/lexical_analysis.html#literals

■ 其他推荐的书

关于制作编程语言的书虽然经常会出，但是多数由于出版量不大而慢慢地绝版了……（希望本书不会这样）

因此我将以前学习过的书作一个介绍（说句题外话，近藤嘉雪老师的“yacc によるC コンパイラプログラミング”（《使用 yacc 开发 C 语言编译器》）一书在日本亚马逊网站上卖到了 245 000 日元[①]，即使是这样我仍然觉得这本书很值得），在这里只介绍一些价格合适并且在市面上可以买到的书。

下面列出的是本书出版时（2009 年 4 月）的参考书籍。

● 新コンピュータサイエンス講義 コンパイラ

中文译名：《新计算机科学讲座：编译器》

作者：田中育男

出版社：Ohmsha 出版社，1995

日本编译器第一人田中育男先生的书。

田中先生的这本书以理论为主，难点很多。书中记录了类似于 Pascal 的语言处理“PL/0”的全部代码，很有实用价值。PL/0 是一个递归下降语法分析器，因此本书可以为不想使用 yacc 来制作编程语言的人提供参考。

● lex & yacc プログラミング

英文版：*lex & yacc*

中文版：《lex 与 yacc》（机械工业出版社 已绝版）

O’Reilly 的动物系列图书。我认为就凭它在这个系列中，这本书就值得信赖。它的出版时间较早，但可以作为 yacc/lex 的参考手册，是一本非常有实用价值的书。

下面这些书是我目前正在学习的。

● コンパイラⅠ－原理· 技法· ツール

英文版：*Compilers: Principles, Techniques, and Tools (2nd Edition)*

中文版：《编译原理》（机械工业出版社）

可以说是编译器制作方面的圣经之著。因为封面上印有龙的图案所以被叫作“龙书”（确切地说应该是“红龙书”）（第 4 章中译者也引用了这本书中的内容）。

本书是原版的第一卷。现在，第二卷已经很难见到了（即便是作者这样的资深人士也没有机会收藏）。

本书内容不太容易理解（例如在不使用 yacc 的情况下制作 LR 解析器），但

① 相当于人民币 15 000 元左右。——译者注

我觉得这是本不可不读的好书。

● コンパイラの構成と最適化

中文译名:《编译器的结构与优化》

作者：田中育男

出版社：朝仓书店，1999

又是田中先生的书。其中对“优化”的 讲解占到了本书一半以上的篇幅。

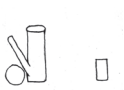
如果你想要了解现在的商业化编译器是怎样做的，这本书再合适不过了。

● *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*

作者：Richar Jones,Rafael Lins

出版社：John Wiley & Sons，1996

尚无译本。本书中不仅介绍了 Mark-Sweep GC 和 Copying GC 等算法，还讲解了通过简单地实现来解决问题的方法（世代型 GC、增量 GC、并发 GC 等）。



图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
分类建议 计算机/编程语言
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33320-9

ISBN 978-7-115-33320-9
定价：79.00元